Table of Contents

A Byte of Python

"A Byte of Python" is a free book on programming using the Python language. It serves as a tutorial or guide to the Python language for a beginner audience. If all you know about computers is how to save text files, then this is the book for you.

For Python version 3

This book will teach you to use Python version 3. There will also be guidance for you to adapt to the older and more common Python version 2 in the book.

Who reads A Byte of Python?

Here are what people are saying about the book:

> This is the best beginner's tutorial I've ever seen! Thank you for your effort. -- Walt Michalik

> The best thing i found was "A Byte of Python", which is simply a brilliant book for a beginner. It's well written, the concepts are well explained with self evident examples. -- Joshua Robin

> Excellent gentle introduction to programming #Python for beginners -- Shan Rajasekaran

> Best newbie guide to python -- Nickson Kaigi

> start to love python with every single page read -- Herbert Feutl

> perfect beginners guide for python, will give u key to unlock magical world of python -- Dilip

> I should be doing my actual "work" but just found "A Byte of Python". A great guide with great examples. -- Biologist John

> Recently started reading a Byte of python. Awesome work. And that too for free. Highly recommended for aspiring pythonistas. -- Mangesh

> A Byte of Python, written by Swaroop. (this is the book I'm currently reading). Probably the best to start with, and probably the best in the world for every newbie or even a more experienced user. -- Apostolos

> Enjoying Reading #ByteOfPython by @swaroopch best book ever -- Yuvraj Sharma

> Thank you so much for writing A Byte Of Python. I just started learning how to code two days ago and I'm already building some simple games. Your guide has been a dream and I just wanted to let you know how valuable it has been. -- Franklin

> I'm from Dayanandasagar College of Engineering (7th sem, CSE). Firstly i want to say that your book "The byte of python" is too good a book for a beginner in python like me.The concepts are so well explained with simple examples that helped me to easily learn python. Thank you so much. -- Madhura

> I am a 18 year old IT student studying at University in Ireland. I would like to express my gratitude to you for writing your book "A Byte of Python", I already had knowledge of 3 programming langagues - C, Java and Javascript, and Python was by far the easiest langague I have ever learned, and that was mainly because your book was fantastic and made learning python very simple and interesting. It is one of the best written and easy to follow programming books I have ever read. Congratulations and keep up the great work. -- Matt

> Hi, I'm from Dominican Republic. My name is Pavel, recently I read your book *A Byte of Python* and I consider it excellent!! :). I learnt much from all the examples. Your book is of great help for newbies like me... -- Pavel Simo

> I am a student from China, Now ,I have read you book A byte of Python, Oh it's beautiful. The book is very simple but can help all the first learnners. You know I am interesting in Java and cloud computing many times, i have to coding programm for the server, so i think python is a good choice, finish your book, i think its not only a good choice its must use the Python. My English is not very well, the email to you, i just wanna thank you! Best Wishes for you and your family. -- Roy Lau

> I recently finished reading Byte of Python, and I thought I really ought to thank you. I was very sad to reach the final pages as I now have to go back to dull, tedious oreilly or etc. manuals for learning about python. Anyway, I really appreciate your book. Samuel Young

> Dear Swaroop, I am taking a class from an instructor that has no interest in teaching. We are using Learning Python, second edition, by O'Reilly. It is not a text for beginner without any programming knowledge, and an instructor that should be working in another field. Thank you very much for your book, without it I would be clueless about Python and programming. Thanks a million, you are able to *break the message down* to a level that beginners can understand and not everyone can. -- Joseph Duarte

> I love your book! It is the greatest Python tutorial ever, and a very useful reference. Brilliant, a true masterpiece! Keep up the good work! -- Chris-André Sommerseth

> First of all, I want to say thanks to you for this greate book. I think it is a good book for those who are looking for a beginner's tutorial for Python. It is about two or there years ago, I think, when I first heard of this book. At that time, I am not able to read some book in English yet, so I got a chinese translation, which took me into the gate of Python programming. Recently, I reread this book. This time, of course, the english version. I couldn't believe that I can read the whole book without my dictionary at hand. Of course, it all dues to your effort to make this book an easy-to-understand one. -- myd7349

> I'm just e-mailing you to thank you for writing Byte of Python online. I had been attempting Python for a few months prior to stumbling across your book, and although I made limited success with pyGame, I never completed a program. Thanks to your simplification of the categories, Python actually seems a reachable goal. It seems like I have finally learned the foundations and I can continue into my real goal, game development. ... Once again, thanks VERY much for placing such a structured and helpful guide to basic programming on the web. It shoved me into and out of OOP with an understanding where two text books had failed. -- Matt Gallivan

> I would like to thank you for your book *A Byte of Python* which i myself find the best way to learn python. I am a 15 year old i live in egypt my name is Ahmed. Python was my second programming language i learn visual basic 6 at school but didn't enjoy it, however i really enjoyed learning python. I made the addressbook program and i was sucessful. i will try to start make more programs and read python programs (if you could tell me source that would be helpful). I will also start on learning java and if you can tell me where to find a tutorial as good as yours for java that would help me a lot. Thanx. -- Ahmed Mohammed

> A wonderful resource for beginners wanting to learn more about Python is the 110-page PDF tutorial A Byte of Python by Swaroop C H. It is well-written, easy to follow, and may be the best introduction to Python programming available. -- Drew Ames

> Yesterday I got through most of Byte of Python on my Nokia N800 and it's the easiest and most concise introduction to Python I have yet encountered. Highly recommended as a starting point for learning Python. -- Jason Delport

> Byte of Vim and Python by @swaroopch is by far the best works in technical writing to me. Excellent reads #FeelGoodFactor -- Surendran

> "Byte of python" best one by far man (in response to the question "Can anyone suggest a good, inexpensive resource for learning the basics of Python? ") -- Justin LoveTrue

> The Book Byte of python was very helpful ..Thanks bigtime :) Chinmay

> Always been a fan of A Byte of Python - made for both new and experienced programmers. -- Patrick Harrington

> I started learning python few days ago from your book..thanks for such a nice book. it is so well written, you made my life easy..so you found a new fan of yours..thats me :) tons of thanks. -- Gadadhari Bheem

> Before I started to learn Python, I've acquired basic programming skills in Assembly, C, C++, C# and Java. The very reason I wanted to learn Python is it's popular (people are talking about it) and powerful (reality). This book written by Mr. Swaroop is a very good guide for both brand-new programmers and new python programmers. Took 10 half days to go through it. Great Help! -- Fang Biyi (PhD Candidate ECE, Michigan State University)

> Thank you ever so much for this book!! This book cleared up many questions I had about certain aspects of Python such as object oriented programming. I do not feel like an expert at OO but I know this book helped me on a first step or two. I have now written several python programs that actually do real things for me as a system administrator. They are all procedural oriented but they are small by most peoples standards. Again, thanks for this book. Thank you for having it on the web. -- Bob

> I just want to thank you for writing the first book on programming I've ever really read. Python is now my first language, and I can just imagine all the possibilities. So thank you for giving me the tools to create things I never would have imagined I could do before. -- "The Walrus"

> I wanted to thank you for writing *A Byte Of Python* (2 & 3 Versions). It has been invaluable to my learning experience in Python & Programming in general. Needless to say, I am a beginner in the programming world, a couple of months of self study up to this point. I had been using youtube tutorials & some other online tutorials including other free books. I decided to dig into your book yesterday, & I've learned more on the first few pages than any other book or tutorial. A few things I had been confused about, were cleared right up with a GREAT example & explanation. Can't wait to read (and learn) more!! Thank you so much for not only writing the book, but for putting it under the creative commons license (free). Thank goodness there are unselfish people like you out there to help & teach the rest of us. -- Chris

> I wrote you back in 2011 and I was just getting into Python and wanted to thank you for your tutorial "A Byte of Python". Without it, I would have fallen by the wayside. Since then I have gone on to program a number of functions in my organization with this language with yet more on the horizon. I would not call myself an advanced programmer by any stretch but I notice the occasional request for assistance now from others since I started using it. I discovered, while reading "Byte" why I had ceased studying C and C++ and it was because the book given to me started out with an example containing an augmented assignment. Of course, there was no explanation for this arrangement of operators and I fell on my head trying to make sense of what was on the written page. As I recall it was a most frustrating exercise which I eventually abandoned. Doesn't mean C or C++ is impossible to learn, or even that I am stupid, but it does mean that the documentation I worked my way through did not define the symbols and words which is an essential part of any instruction. Just as computers will not be able to understand a computer word or computer symbol that is outside the syntax for the language being used, a student new to any field will not grasp his subject if he encounters words or symbols for which there are no definitions. You get a "blue screen" as it were in either case. The solution is simple, though: find the word or symbol and get the proper definition or symbol and lo and behold,the computer or student can proceed. Your book was so well put together that I found very little in it I couldn't grasp. So, thank you. I encourage you to continue to include full definitions of terms. The documentation with Python is good, once you know, (the examples are its strength from what I see) but in many cases it seems that you have to know in order to understand the documentation which to my mind is not what should be. Third party tutorials express the need for clarification of the documentation and their success largely depends on the words that are used to describe the terminology. I have recommended your book to many others. Some in Australia, some in the Caribbean and yet others in the US. It fills a niche no others do. I hope you are doing well and wish you all the success in the future. -- Nick

> hey, this is ankush(19). I was facing a great difficulty to start with python. I tried a lot of books but all were bulkier and not target oriented; and then i found this lovely one, which made me love python in no time. Thanks a lot for this "beautiful piece of book". -- Ankush

> I would like to thank you for your excellent guide on Python. I am a molecular biologist (with little programming background) and for my work I need to handle big datasets of DNA sequences and to analyse microscope images. For both things, programming in python has been useful, if not essential to complete and publish a 6-years project. That such a guide is freely available is a clear sign that the forces of evil are not yet ruling the world! :) -- Luca

> Since this is going to be the first language you learn, you should use A Byte of Python. It really gives a proper introduction into programming in Python and it is paced well enough for the average beginner. The most important thing from then on will be actually starting to practice making your own little programs. -- "{Unregistered}"

> Just to say a loud and happy *thank you very much* for publishing "A Byte of Python" and "A Byte of Vim". Those books were very useful to me four or five years ago when I starting learning programming. Right now I'm developing a project that was a dream for a long, long time and just want to say *thank you*. Keep walking. You are a source of motivation. All the best. -- Jocimar

> Finished reading A byte of Python in 3 days. It is thoroughly interesting. Not a single page was boring. I want to understand the Orca screen reader code. Your book has hopefully equipped me for it. -- Dattatray

> Hi, 'A byte of python' is really a good reading for python beginners. So, again, NICE WORK! i'm a 4 years experienced Java&C developer from China. Recently, i want to do some work on zim-wiki note project which uses pygtk to implement. i read your book in 6 days, and i can read and write python code examples now. thx for your contribution. plz keep your enthusiasm to make this world better, this is just a little encourage from China. -- Lee

> I am Isen from Taiwan, who is a graduating PhD student in Electrical Engineering Department of National Taiwan University. I would like to thank you for your great book. I think it is not only just easy to read but also comprehensive and complete for a new comer of Python. The reason I read your book is that I am starting to work on the GNU Radio framework. Your book let me catch most of important core ideas and skill of Python with a minimum time. I also saw that you do not mind that readers send you a thank note in your book. So I really like your book and appreciate it. Thanks. -- Isen I-Chun Chao

The book is even used by NASA! It is used in their Jet Propulsion Laboratory with their Deep Space Network project.

Academic Courses

This book is/was being used as instructional material in various educational institutions:

- 'Principles of Programming Languages' course at Vrije Universiteit, Amsterdam
- 'Basic Concepts of Computing' course at University of California, Davis
- 'Programming With Python' course at Harvard University
- 'Introduction to Programming' course at University of Leeds

- 'Introduction to Application Programming' course at Boston University
- 'Information Technology Skills for Meteorology' course at University of Oklahoma
- 'Geoprocessing' course at Michigan State University
- 'Multi Agent Semantic Web Systems' course at the University of Edinburgh
- 'Introduction to Computer Science and Programming' at MIT OpenCourseWare
- 'Basic programming at the Faculty of Social Sciences, University of Ljubljana, Slovenia' -- Aleš Žiberna says *"I (and my predecessor) have been using your book as the main literature for this course"*
- 'Introduction to programming', Department of Information Sciences, University of Zadar, Croatia -- Krešimir Zauder says *"I would like to inform you that A Byte of Python is a mandatory read at my course"*

License

This book is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

This means:

- You are free to Share i.e. to copy, distribute and transmit this book
- You are free to Remix i.e. to make changes to this book (especially translations)
- You are free to use it for commercial purposes

Please note:

- Please do *not* sell electronic or printed copies of the book unless you have clearly and prominently mentioned in the description that these copies are *not* from the original author of this book.
- Attribution *must* be shown in the introductory description and front page of the document by linking back to https://python.swaroopch.com/ and clearly indicating that the original text can be fetched from this location.
- All the code/scripts provided in this book is licensed under the 3-clause BSD License unless otherwise noted.

Read Now

You can read it online at https://python.swaroopch.com/

Buy The Book

A printed hardcopy of the book can be purchased at https://www.swaroopch.com/buybook/ for your offline reading pleasure, and to support the continued development and improvement of this book.

Download

Visit https://www.gitbook.com/book/swaroopch/byte-of-python/details for the following types of downloads:

- PDF (for desktop reading, etc.)
- EPUB (for iPhone/iPad, ebook readers, etc.)
- Mobi (for Kindle)

Visit https://github.com/swaroopch/byte-of-python for the raw content (for suggesting corrections, changes, translating, etc.)

Read the book in your native language

If you are interested in reading or contributing translations of this book to other human languages, please see Translations.

Dedication

To Kalyan Varma and many other seniors at PESIT who introduced us to GNU/Linux and the world of open source.

To the memory of Atul Chitnis, a friend and guide who shall be missed greatly.

To the pioneers who made the Internet happen. This book was first written in 2003. It still remains popular, thanks to the nature of sharing knowledge on the Internet as envisioned by the pioneers.

Preface

Python is probably one of the few programming languages which is both simple and powerful. This is good for beginners as well as for experts, and more importantly, is fun to program with. This book aims to help you learn this wonderful language and show how to get things done quickly and painlessly - in effect 'The Anti-venom to your programming problems'.

Who This Book Is For

This book serves as a guide or tutorial to the Python programming language. It is mainly targeted at newbies. It is useful for experienced programmers as well.

The aim is that if all you know about computers is how to save text files, then you can learn Python from this book. If you have previous programming experience, then you can also learn Python from this book.

If you do have previous programming experience, you will be interested in the differences between Python and your favorite programming language - I have highlighted many such differences. A little warning though, Python is soon going to become your favorite programming language!

Official Website

The official website of the book is https://python.swaroopch.com/ where you can read the whole book online, download the latest versions of the book, buy a printed hard copy and also send me feedback.

Something To Think About

> There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies. -- C. A. R. Hoare

> Success in life is a matter not so much of talent and opportunity as of concentration and perseverance. -- C. W. Wendte

About Python

Python is one of those rare languages which can claim to be both *simple* and *powerful*. You will find yourself pleasantly surprised to see how easy it is to concentrate on the solution to the problem rather than the syntax and structure of the language you are programming in.

The official introduction to Python is:

> Python is an easy to learn, powerful programming language. It has efficient high-level data structures and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

I will discuss most of these features in more detail in the next section.

Story behind the name

Guido van Rossum, the creator of the Python language, named the language after the BBC show "Monty Python's Flying Circus". He doesn't particularly like snakes that kill animals for food by winding their long bodies around them and crushing them.

Features of Python

Simple

Python is a simple and minimalistic language. Reading a good Python program feels almost like reading English, although very strict English! This pseudo-code nature of Python is one of its greatest strengths. It allows you to concentrate on the solution to the problem rather than the language itself.

Easy to Learn

As you will see, Python is extremely easy to get started with. Python has an extraordinarily simple syntax, as already mentioned.

Free and Open Source

Python is an example of a *FLOSS* (Free/Libré and Open Source Software). In simple terms, you can freely distribute copies of this software, read its source code, make changes to it, and use pieces of it in new free programs. FLOSS is based on the concept of a community which shares knowledge. This is one of the reasons why Python is so good - it has been created and is constantly improved by a community who just want to see a better Python.

High-level Language

When you write programs in Python, you never need to bother about the low-level details such as managing the memory used by your program, etc.

Portable

Due to its open-source nature, Python has been ported to (i.e. changed to make it work on) many platforms. All your Python programs can work on any of these platforms without requiring any changes at all if you are careful enough to avoid any system-dependent features.

You can use Python on GNU/Linux, Windows, FreeBSD, Macintosh, Solaris, OS/2, Amiga, AROS, AS/400, BeOS, OS/390, z/OS, Palm OS, QNX, VMS, Psion, Acorn RISC OS, VxWorks, PlayStation, Sharp Zaurus, Windows CE and PocketPC!

You can even use a platform like Kivy to create games for your computer *and* for iPhone, iPad, and Android.

Interpreted

This requires a bit of explanation.

A program written in a compiled language like C or C++ is converted from the source language i.e. C or C++ into a language that is spoken by your computer (binary code i.e. 0s and 1s) using a compiler with various flags and options. When you run the program, the linker/loader software copies the program from hard disk to memory and starts running it.

Python, on the other hand, does not need compilation to binary. You just *run* the program directly from the source code. Internally, Python converts the source code into an intermediate form called bytecodes and then translates this into the native language of your computer and then runs it. All this, actually, makes using Python much easier since you don't have to worry about compiling the program, making sure that the proper libraries are linked and loaded, etc. This also makes your Python programs much more portable, since you can just copy your Python program onto another computer and it just works!

Object Oriented

Python supports procedure-oriented programming as well as object-oriented programming. In *procedure-oriented* languages, the program is built around procedures or functions which are nothing but reusable pieces of programs. In *object-oriented* languages, the program is built around objects which combine data and functionality. Python has a very powerful but simplistic way of doing OOP, especially when compared to big languages like C++ or Java.

Extensible

If you need a critical piece of code to run very fast or want to have some piece of algorithm not to be open, you can code that part of your program in C or C++ and then use it from your Python program.

Embeddable

You can embed Python within your C/C++ programs to give *scripting* capabilities for your program's users.

Extensive Libraries

The Python Standard Library is huge indeed. It can help you do various things involving regular expressions,documentation generation, unit testing, threading, databases, web browsers, CGI, FTP, email, XML, XML-RPC, HTML, WAV files, cryptography, GUI (graphical user interfaces), and other system-dependent stuff. Remember, all this is always available wherever Python is installed. This is called the *Batteries Included* philosophy of Python.

Besides the standard library, there are various other high-quality libraries which you can find at the Python Package Index.

Summary

Python is indeed an exciting and powerful language. It has the right combination of performance and features that make writing programs in Python both fun and easy.

Python 3 versus 2

You can ignore this section if you're not interested in the difference between "Python version 2" and "Python version 3". But please do be aware of which version you are using. This book is written for Python version 3.

Remember that once you have properly understood and learn to use one version, you can easily learn the differences and use the other one. The hard part is learning programming and understanding the basics of Python language itself. That is our goal in this book, and once you have achieved that goal, you can easily use Python 2 or Python 3 depending on your situation.

For details on differences between Python 2 and Python 3, see:

- The future of Python 2
- Porting Python 2 Code to Python 3
- Writing code that runs under both Python2 and 3
- Supporting Python 3: An in-depth guide

What Programmers Say

You may find it interesting to read what great hackers like ESR have to say about Python:

- *Eric S. Raymond* is the author of "The Cathedral and the Bazaar" and is also the person who coined the term *Open Source*. He says that Python has become his favorite programming language. This article was the real inspiration for my first brush with Python.
- *Bruce Eckel* is the author of the famous 'Thinking in Java' and 'Thinking in C++' books. He says that no language has made him more productive than Python. He says that Python is perhaps the only language that focuses on making things easier for the programmer. Read the complete interview for more details.
- *Peter Norvig* is a well-known Lisp author and Director of Search Quality at Google (thanks to Guido van Rossum for pointing that out). He says that writing Python is like writing in pseudocode. He says that Python has always been an integral part of Google. You can actually verify this statement by looking at the Google Jobs page which lists Python knowledge as a requirement for software engineers.

Installation

When we refer to "Python 3" in this book, we will be referring to any version of Python equal to or greater than version Python 3.6.0.

Installation on Windows

Visit https://www.python.org/downloads/ and download the latest version. At the time of this writing, it was Python 3.5.1 The installation is just like any other Windows-based software.

Note that if your Windows version is pre-Vista, you should download Python 3.4 only as later versions require newer versions of Windows.

CAUTION: Make sure you check option `Add Python 3.5 to PATH` .

To change install location, click on `Customize installation` , then `Next` and enter `C:\python35` (or another appropriate location) as the install location.

If you didn�t check the `Add Python 3.5 PATH` option earlier, check `Add Python to environment variables` . This does the same thing as `Add Python 3.5 to PATH` on the first install screen.

You can choose to install Launcher for all users or not, it does not matter much. Launcher is used to switch between different versions of Python installed.

If your path was not set correctly (by checking the `Add Python 3.5 Path` or `Add Python to environment variables` options), then follow the steps in the next section (`DOS Prompt`) to fix it. Otherwise, go to the `Running Python prompt on Windows` section in this document.

NOTE: For people who already know programming, if you are familiar with Docker, check out Python in Docker and Docker on Windows.

DOS Prompt

If you want to be able to use Python from the Windows command line i.e. the DOS prompt, then you need to set the PATH variable appropriately.

For Windows 2000, XP, 2003 , click on `Control Panel` -> `System` -> `Advanced` -> `Environment Variables` . Click on the variable named `PATH` in the *System Variables* section, then select `Edit` and add `;C:\Python35` (please verify that this folder exists, it will be

different for newer versions of Python) to the end of what is already there. Of course, use the appropriate directory name.

For older versions of Windows, open the file `C:\AUTOEXEC.BAT` and add the line `PATH=%PATH%;C:\Python35` and restart the system. For Windows NT, use the `AUTOEXEC.NT` file.

For Windows Vista:

- Click Start and choose `Control Panel`
- Click System, on the right you'll see "View basic information about your computer"
- On the left is a list of tasks, the last of which is `Advanced system settings`. Click that.
- The `Advanced` tab of the `System Properties` dialog box is shown. Click the `Environment Variables` button on the bottom right.
- In the lower box titled `System Variables` scroll down to Path and click the `Edit` button.
- Change your path as need be.
- Restart your system. Vista didn't pick up the system path environment variable change until I restarted.

For Windows 7 and 8:

- Right click on Computer from your desktop and select `Properties` or click `Start` and choose `Control Panel` -> `System and Security` -> `System`. Click on `Advanced system settings` on the left and then click on the `Advanced` tab. At the bottom click on `Environment Variables` and under `System variables`, look for the `PATH` variable, select and then press `Edit`.
- Go to the end of the line under Variable value and append `;C:\Python35` (please verify that this folder exists, it will be different for newer versions of Python) to the end of what is already there. Of course, use the appropriate folder name.
- If the value was `%SystemRoot%\system32;` It will now become `%SystemRoot%\system32;C:\Python36`
- Click `OK` and you are done. No restart is required, however you may have to close and reopen the command line.

For Windows 10:

Windows Start Menu > `Settings` > `About` > `System Info` (this is all the way over to the right) > `Advanced System Settings` > `Environment Variables` (this is towards the bottom) > (then highlight `Path` variable and click `Edit`) > `New` > (type in whatever your python location is. For example, `C:\Python35\`)

Running Python prompt on Windows

For Windows users, you can run the interpreter in the command line if you have set the `PATH` variable appropriately.

To open the terminal in Windows, click the start button and click `Run`. In the dialog box, type `cmd` and press `[enter]` key.

Then, type `python` and ensure there are no errors.

Installation on Mac OS X

For Mac OS X users, use Homebrew: `brew install python3`.

To verify, open the terminal by pressing `[Command + Space]` keys (to open Spotlight search), type `Terminal` and press `[enter]` key. Now, run `python3` and ensure there are no errors.

Installation on GNU/Linux

For GNU/Linux users, use your distribution's package manager to install Python 3, e.g. on Debian & Ubuntu: `sudo apt-get update && sudo apt-get install python3`.

To verify, open the terminal by opening the `Terminal` application or by pressing `Alt + F2` and entering `gnome-terminal`. If that doesn't work, please refer the documentation of your particular GNU/Linux distribution. Now, run `python3` and ensure there are no errors.

You can see the version of Python on the screen by running:

```
$ python3 -V
Python 3.6.0
```

NOTE: `$` is the prompt of the shell. It will be different for you depending on the settings of the operating system on your computer, hence I will indicate the prompt by just the `$` symbol.

CAUTION: Output may be different on your computer, depending on the version of Python software installed on your computer.

Summary

From now on, we will assume that you have Python installed on your system.

Next, we will write our first Python program.

First Steps

We will now see how to run a traditional 'Hello World' program in Python. This will teach you how to write, save and run Python programs.

There are two ways of using Python to run your program - using the interactive interpreter prompt or using a source file. We will now see how to use both of these methods.

Using The Interpreter Prompt

Open the terminal in your operating system (as discussed previously in the Installation chapter) and then open the Python prompt by typing `python3` and pressing `[enter]` key.

Once you have started Python, you should see `>>>` where you can start typing stuff. This is called the *Python interpreter prompt*.

At the Python interpreter prompt, type:

```
print("Hello World")
```

followed by the `[enter]` key. You should see the words `Hello World` printed to the screen.

Here is an example of what you should be seeing, when using a Mac OS X computer. The details about the Python software will differ based on your computer, but the part from the prompt (i.e. from `>>>` onwards) should be the same regardless of the operating system.

```
$ python3
Python 3.6.0 (default, Jan 12 2017, 11:26:36)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
```

Notice that Python gives you the output of the line immediately! What you just entered is a single Python *statement*. We use `print` to (unsurprisingly) print any value that you supply to it. Here, we are supplying the text `Hello World` and this is promptly printed to the screen.

How to Quit the Interpreter Prompt

If you are using a GNU/Linux or OS X shell, you can exit the interpreter prompt by pressing `[ctrl + d]` or entering `exit()` (note: remember to include the parentheses, `()`) followed by the `[enter]` key.

If you are using the Windows command prompt, press `[ctrl + z]` followed by the `[enter]` key.

Choosing An Editor

We cannot type out our program at the interpreter prompt every time we want to run something, so we have to save them in files and can run our programs any number of times.

To create our Python source files, we need an editor software where you can type and save. A good programmer's editor will make your life easier in writing the source files. Hence, the choice of an editor is crucial indeed. You have to choose an editor as you would choose a car you would buy. A good editor will help you write Python programs easily, making your journey more comfortable and helps you reach your destination (achieve your goal) in a much faster and safer way.

One of the very basic requirements is *syntax highlighting* where all the different parts of your Python program are colorized so that you can *see* your program and visualize its running.

If you have no idea where to start, I would recommend using PyCharm Educational Edition software which is available on Windows, Mac OS X and GNU/Linux. Details in the next section.

If you are using Windows, *do not use Notepad* - it is a bad choice because it does not do syntax highlighting and also importantly it does not support indentation of the text which is very important in our case as we will see later. Good editors will automatically do this.

If you are an experienced programmer, then you must be already using Vim or Emacs. Needless to say, these are two of the most powerful editors and you will benefit from using them to write your Python programs. I personally use both for most of my programs, and have even written an entire book on Vim.

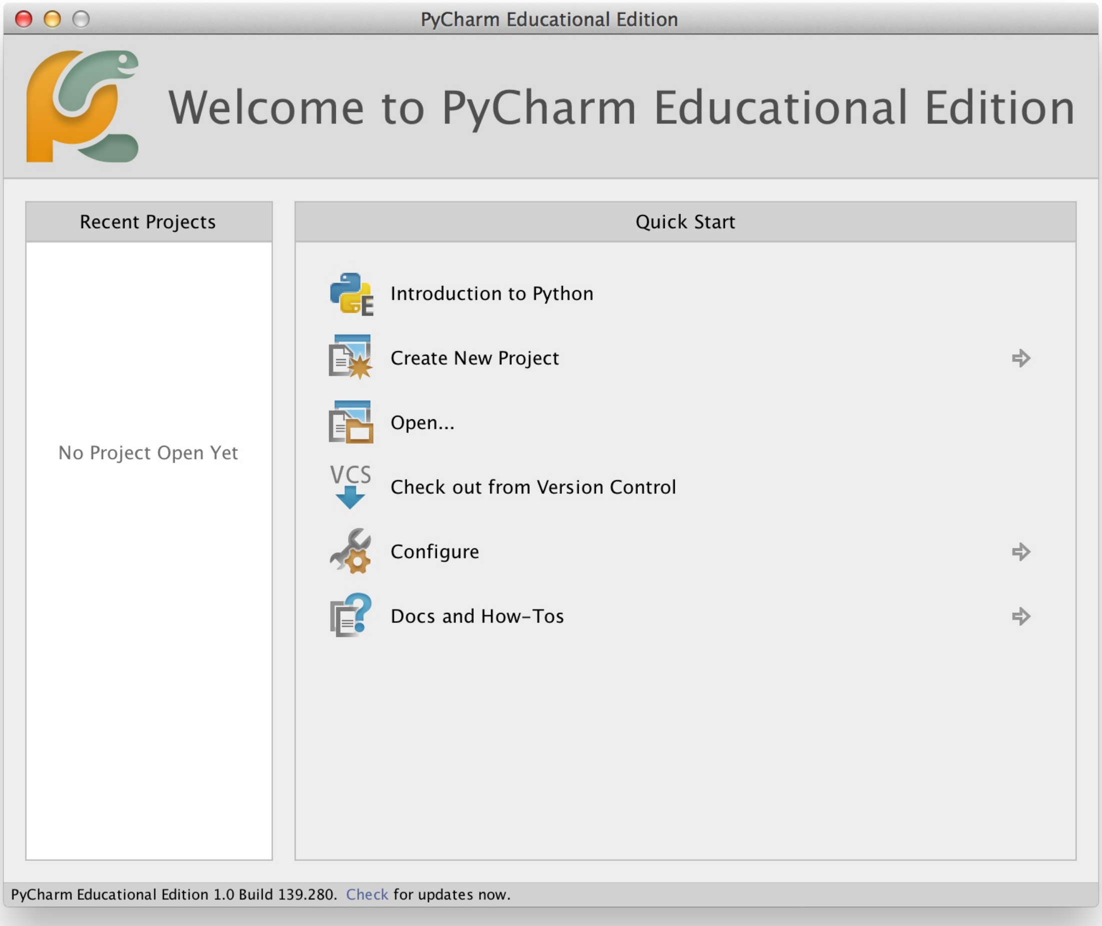
In case you are willing to take the time to learn Vim or Emacs, then I highly recommend that you do learn to use either of them as it will be very useful for you in the long run. However, as I mentioned before, beginners can start with PyCharm and focus the learning on Python rather than the editor at this moment.

To reiterate, please choose a proper editor - it can make writing Python programs more fun and easy.

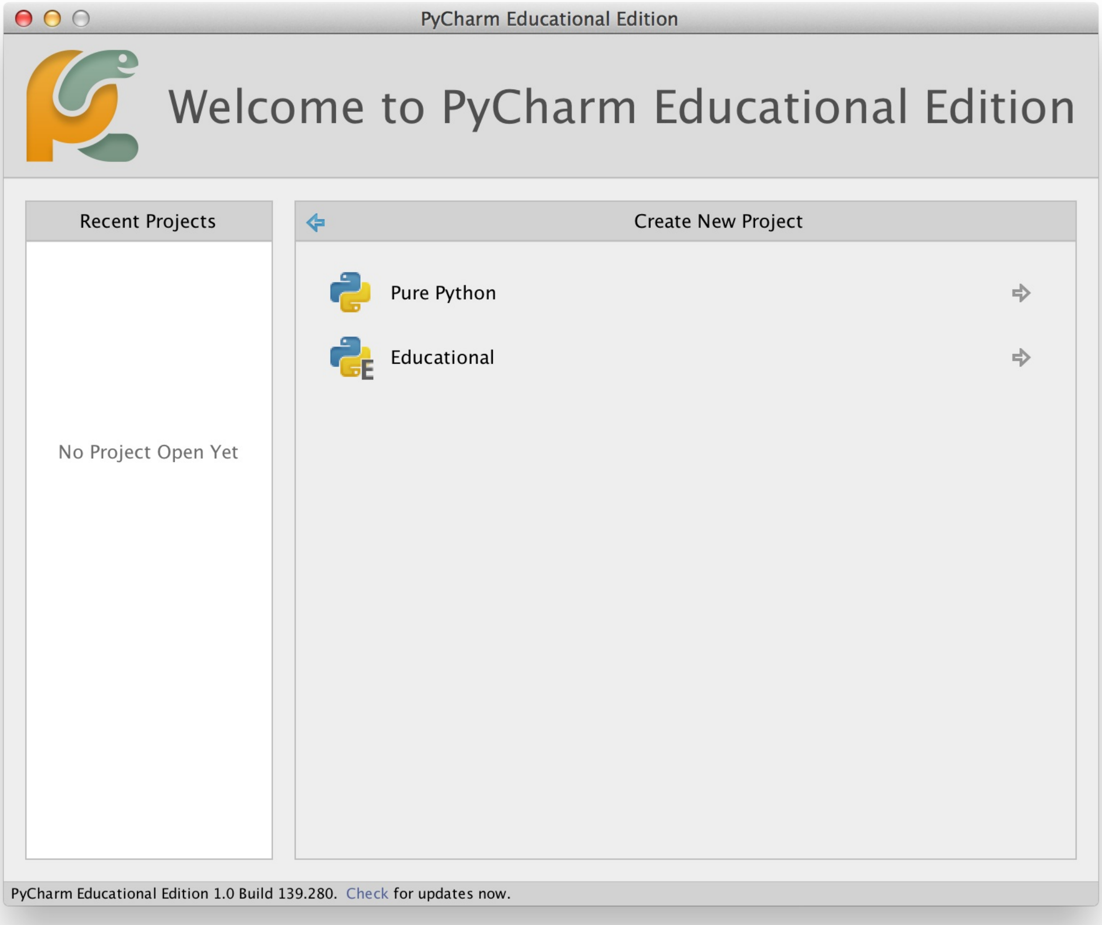
PyCharm

PyCharm Educational Edition is a free editor which you can use for writing Python programs.

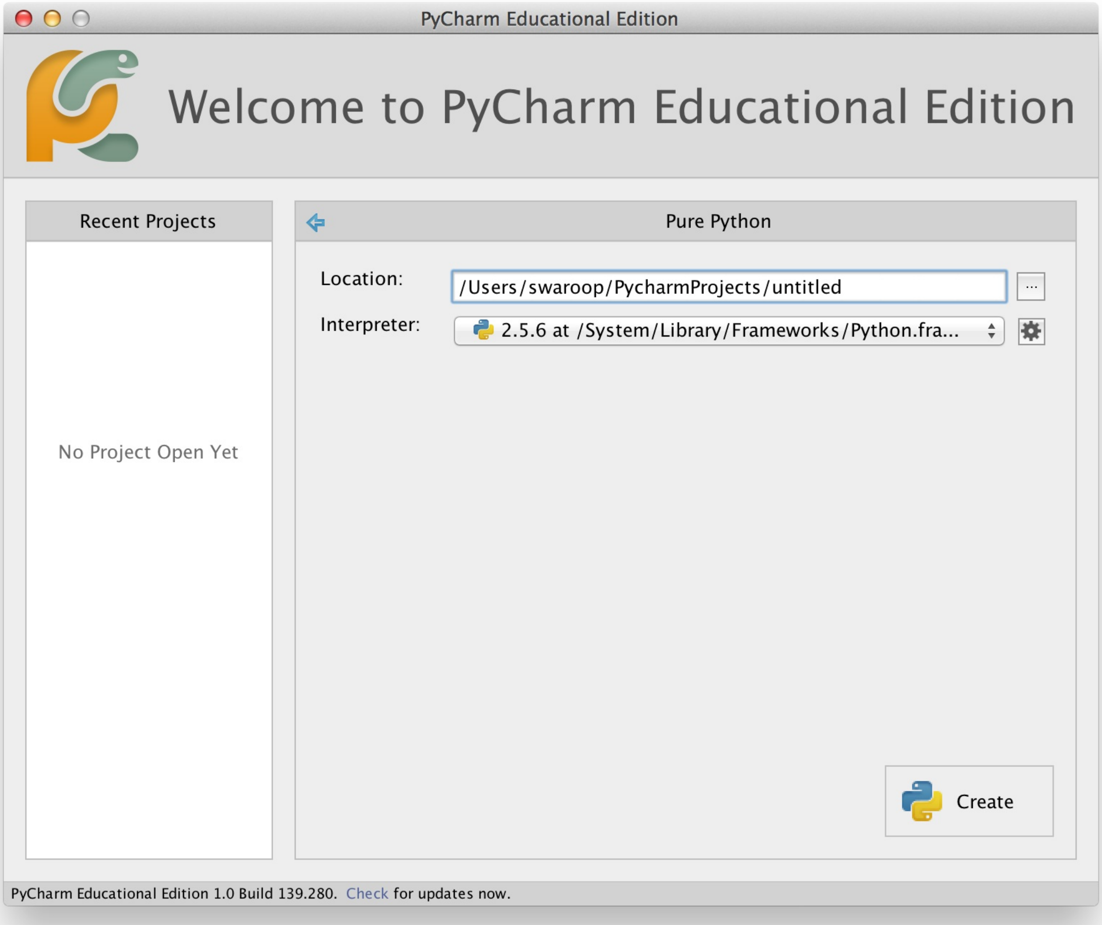
When you open PyCharm, you'll see this, click on `Create New Project` :



Select `Pure Python` :

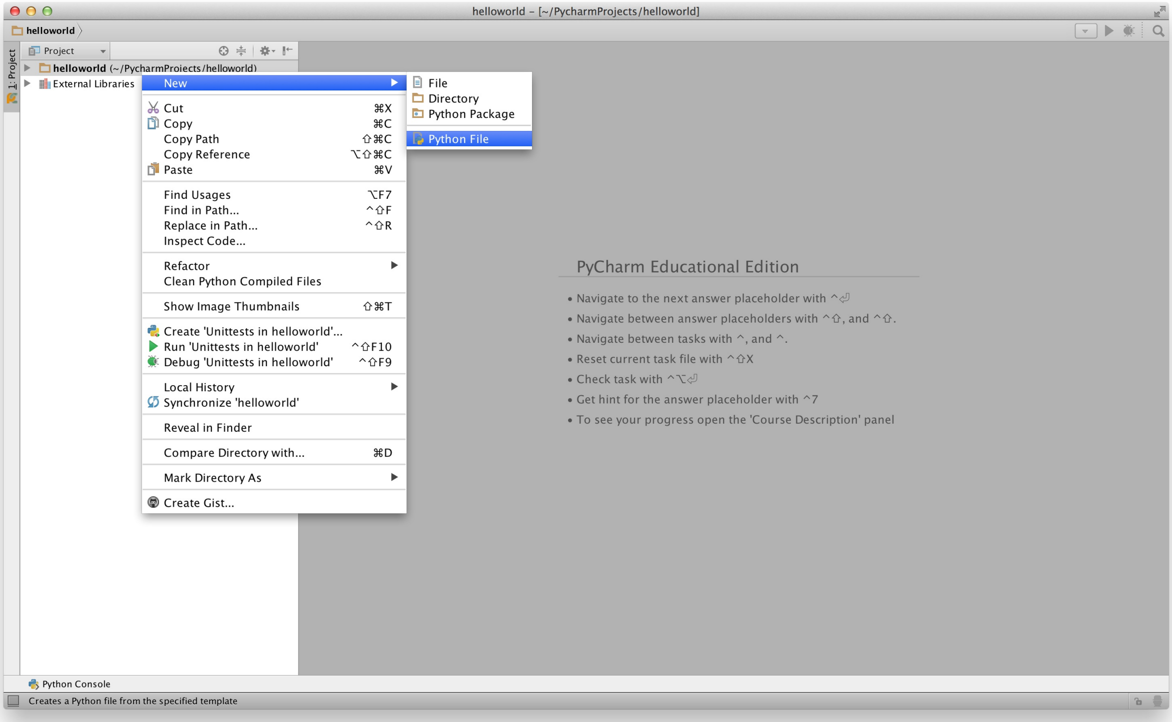


Change `untitled` to `helloworld` as the location of the project, you should see details similar to this:

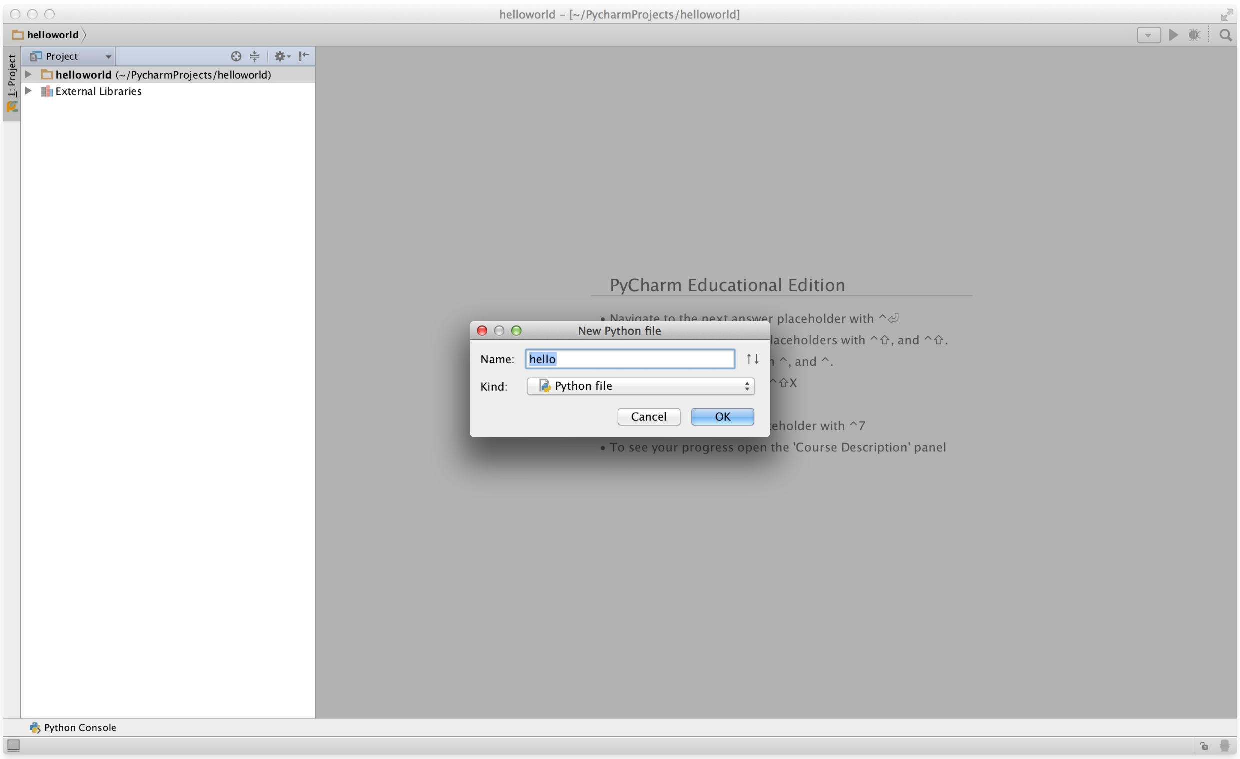


Click the `Create` button.

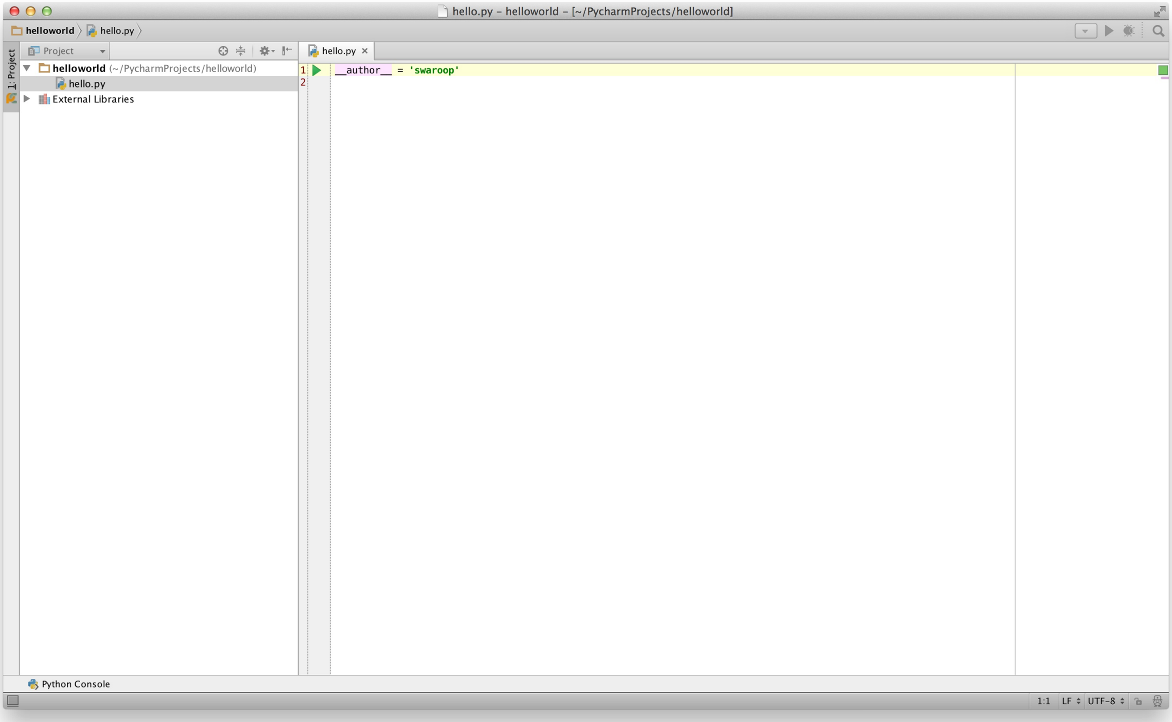
Right-click on the `helloworld` in the sidebar and select `New` -> `Python File` :



You will be asked to type the name, type `hello` :



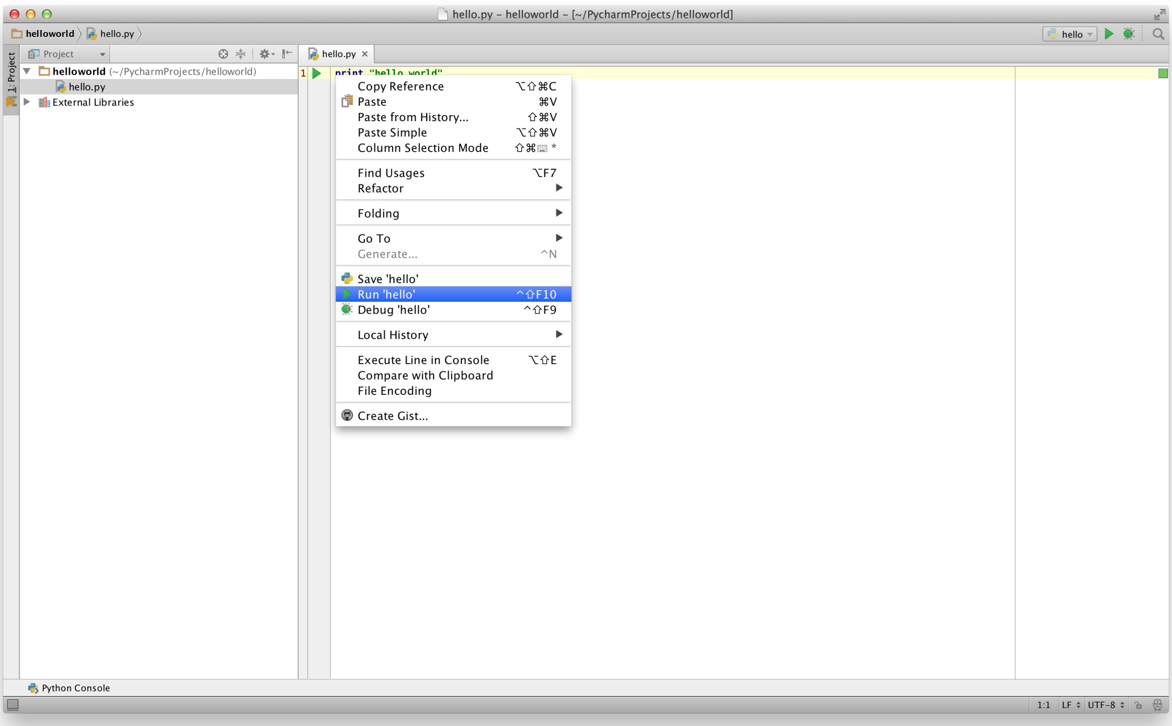
You can now see a file opened for you:



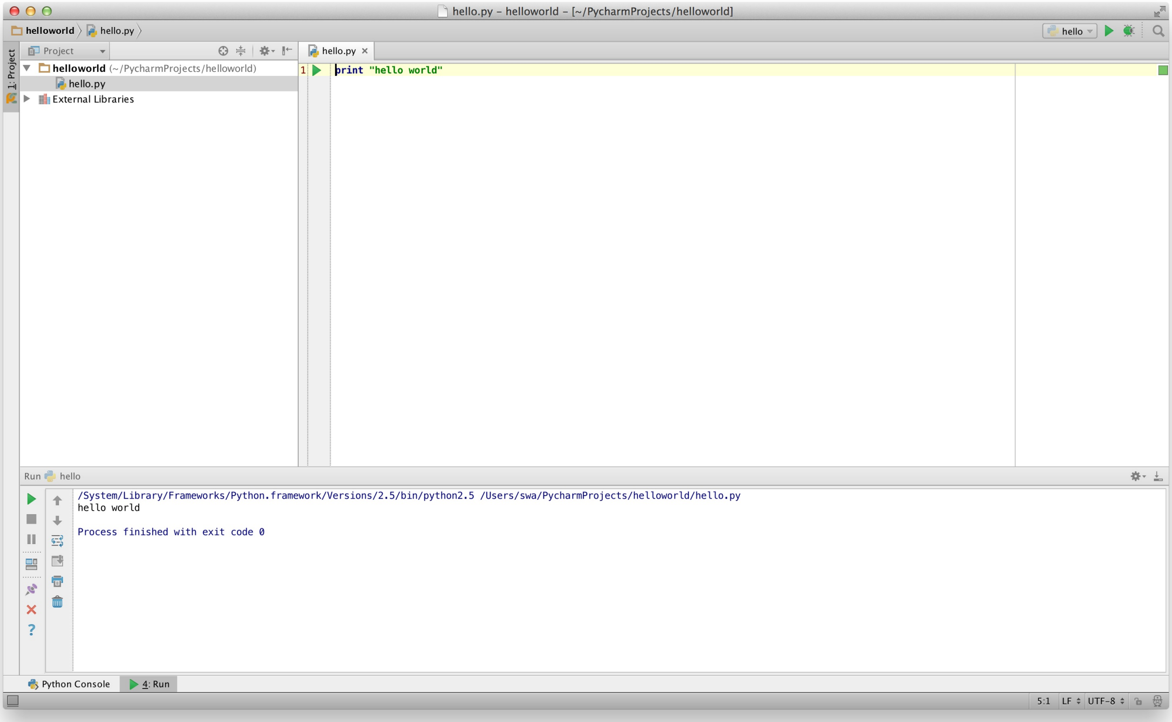
Delete the lines that are already present, and now type the following:

```
print("hello world")
```

Now right-click on what you typed (without selecting the text), and click on `Run 'hello'`.



You should now see the output (what it prints) of your program:



Phew! That was quite a few steps to get started, but henceforth, every time we ask you to create a new file, remember to just right-click on `helloworld` on the left -> `New` -> `Python File` and continue the same steps to type and run as shown above.

You can find more information about PyCharm in the PyCharm Quickstart page.

Vim

1. Install Vim
    - Mac OS X users should install `macvim` package via HomeBrew
    - Windows users should download the "self-installing executable" from Vim website
    - GNU/Linux users should get Vim from their distribution's software repositories, e.g. Debian and Ubuntu users can install the `vim` package.
2. Install jedi-vim plugin for autocompletion.
3. Install corresponding `jedi` python package : `pip install -U jedi`

Emacs

1. Install Emacs 24+.
    - Mac OS X users should get Emacs from http://emacsformacosx.com
    - Windows users should get Emacs from http://ftp.gnu.org/gnu/emacs/windows/

    - GNU/Linux users should get Emacs from their distribution's software repositories, e.g. Debian and Ubuntu users can install the `emacs24` package.
2. Install ELPY

Using A Source File

Now let's get back to programming. There is a tradition that whenever you learn a new programming language, the first program that you write and run is the 'Hello World' program - all it does is just say 'Hello World' when you run it. As Simon Cozens[1] says, it is the "traditional incantation to the programming gods to help you learn the language better."

Start your choice of editor, enter the following program and save it as `hello.py`.

If you are using PyCharm, we have already discussed how to run from a source file.

For other editors, open a new file `hello.py` and type this:

```
print("hello world")
```

Where should you save the file? To any folder for which you know the location of the folder. If you don't understand what that means, create a new folder and use that location to save and run all your Python programs:

- `/tmp/py` on Mac OS X
- `/tmp/py` on GNU/Linux
- `C:\py` on Windows

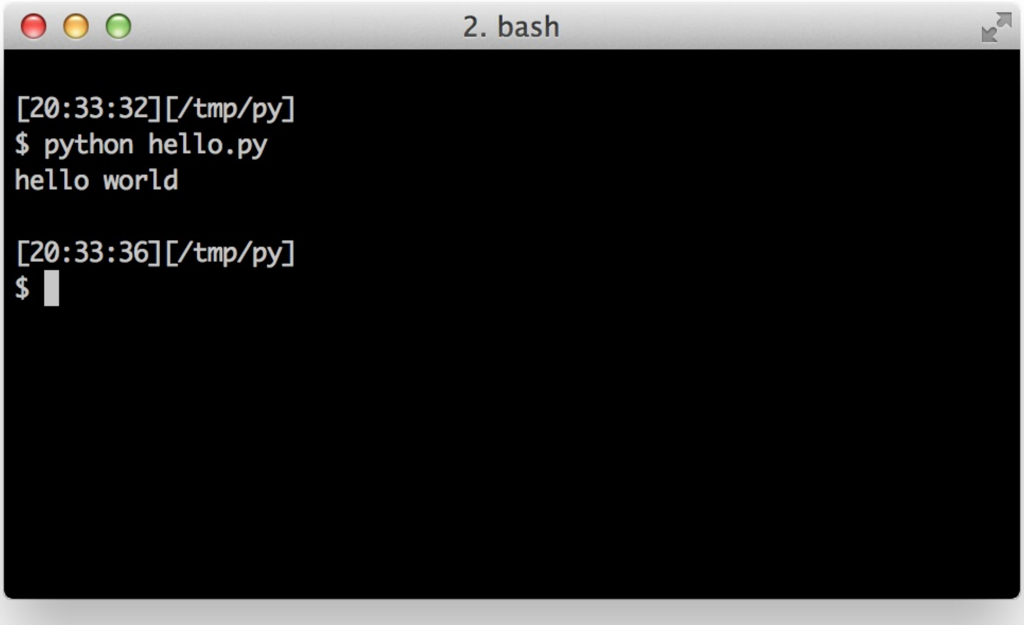
To create the above folder (for the operating system you are using), use the `mkdir` command in the terminal, for example, `mkdir /tmp/py`.

IMPORTANT: Always ensure that you give it the file extension of `.py`, for example, `foo.py`.

To run your Python program:

1. Open a terminal window (see the previous Installation chapter on how to do that)
2. **C**hange **d**irectory to where you saved the file, for example, `cd /tmp/py`
3. Run the program by entering the command `python hello.py`. The output is as shown below.

```
$ python hello.py
hello world
```



If you got the output as shown above, congratulations! - you have successfully run your first Python program. You have successfully crossed the hardest part of learning programming, which is, getting started with your first program!

In case you got an error, please type the above program *exactly* as shown above and run the program again. Note that Python is case-sensitive i.e. `print` is not the same as `Print` - note the lowercase `p` in the former and the uppercase `P` in the latter. Also, ensure there are no spaces or tabs before the first character in each line - we will see why this is important later.

How It Works

A Python program is composed of *statements*. In our first program, we have only one statement. In this statement, we call the `print` *statement* to which we supply the text "hello world".

Getting Help

If you need quick information about any function or statement in Python, then you can use the built-in `help` functionality. This is very useful especially when using the interpreter prompt. For example, run `help('len')` - this displays the help for the `len` function which is used to count number of items.

TIP: Press `q` to exit the help.

Similarly, you can obtain information about almost anything in Python. Use `help()` to learn more about using `help` itself!

In case you need to get help for operators like `return`, then you need to put those inside quotes such as `help('return')` so that Python doesn't get confused on what we're trying to do.

Summary

You should now be able to write, save and run Python programs at ease.

Now that you are a Python user, let's learn some more Python concepts.

> 1. the author of the amazing 'Beginning Perl' book ↩

Basics

Just printing `hello world` is not enough, is it? You want to do more than that - you want to take some input, manipulate it and get something out of it. We can achieve this in Python using constants and variables, and we'll learn some other concepts as well in this chapter.

Comments

Comments are any text to the right of the `#` symbol and is mainly useful as notes for the reader of the program.

For example:

```
print('hello world') # Note that print is a function
```

or:

```
# Note that print is a function
print('hello world')
```

Use as many useful comments as you can in your program to:

- explain assumptions
- explain important decisions
- explain important details
- explain problems you're trying to solve
- explain problems you're trying to overcome in your program, etc.

Code tells you how, comments should tell you why.

This is useful for readers of your program so that they can easily understand what the program is doing. Remember, that person can be yourself after six months!

Literal Constants

An example of a literal constant is a number like `5`, `1.23`, or a string like `'This is a string'` or `"It's a string!"`.

It is called a literal because it is *literal* - you use its value literally. The number `2` always represents itself and nothing else - it is a *constant* because its value cannot be changed. Hence, all these are referred to as literal constants.

Numbers

Numbers are mainly of two types - integers and floats.

An example of an integer is `2` which is just a whole number.

Examples of floating point numbers (or *floats* for short) are `3.23` and `52.3E-4`. The `E` notation indicates powers of 10. In this case, `52.3E-4` means `52.3 * 10^-4^`.

> **Note for Experienced Programmers**
>
> There is no separate `long` type. The `int` type can be an integer of any size.

Strings

A string is a *sequence* of *characters*. Strings are basically just a bunch of words.

You will be using strings in almost every Python program that you write, so pay attention to the following part.

Single Quote

You can specify strings using single quotes such as `'Quote me on this'`.

All white space i.e. spaces and tabs, within the quotes, are preserved as-is.

Double Quotes

Strings in double quotes work exactly the same way as strings in single quotes. An example is `"What's your name?"`.

Triple Quotes

You can specify multi-line strings using triple quotes - (`"""` or `'''`). You can use single quotes and double quotes freely within the triple quotes. An example is:

```
'''This is a multi-line string. This is the first line.
This is the second line.
"What's your name?," I asked.
He said "Bond, James Bond."
'''
```

Strings Are Immutable

This means that once you have created a string, you cannot change it. Although this might seem like a bad thing, it really isn't. We will see why this is not a limitation in the various programs that we see later on.

> **Note for C/C++ Programmers**
>
> There is no separate `char` data type in Python. There is no real need for it and I am sure you won't miss it.

> **Note for Perl/PHP Programmers**
>
> Remember that single-quoted strings and double-quoted strings are the same - they do not differ in any way.

The format method

Sometimes we may want to construct strings from other information. This is where the `format()` method is useful.

Save the following lines as a file `str_format.py` :

```
age = 20
name = 'Swaroop'

print('{0} was {1} years old when he wrote this book'.format(name, age))
print('Why is {0} playing with that python?'.format(name))
```

Output:

```
$ python str_format.py
Swaroop was 20 years old when he wrote this book
Why is Swaroop playing with that python?
```

How It Works

A string can use certain specifications and subsequently, the `format` method can be called to substitute those specifications with corresponding arguments to the `format` method.

Observe the first usage where we use `{0}` and this corresponds to the variable `name` which is the first argument to the format method. Similarly, the second specification is `{1}` corresponding to `age` which is the second argument to the format method. Note that Python starts counting from 0 which means that first position is at index 0, second position is at index 1, and so on.

Notice that we could have achieved the same using string concatenation:

```
name + ' is ' + str(age) + ' years old'
```

but that is much uglier and error-prone. Second, the conversion to string would be done automatically by the `format` method instead of the explicit conversion to strings needed in this case. Third, when using the `format` method, we can change the message without having to deal with the variables used and vice-versa.

Also note that the numbers are optional, so you could have also written as:

```
age = 20
name = 'Swaroop'

print('{} was {} years old when he wrote this book'.format(name, age))
print('Why is {} playing with that python?'.format(name))
```

which will give the same exact output as the previous program.

What Python does in the `format` method is that it substitutes each argument value into the place of the specification. There can be more detailed specifications such as:

```
# decimal (.) precision of 3 for float '0.333'
print('{0:.3f}'.format(1.0/3))
# fill with underscores (_) with the text centered
# (^) to 11 width '___hello___'
print('{0:_^11}'.format('hello'))
# keyword-based 'Swaroop wrote A Byte of Python'
print('{name} wrote {book}'.format(name='Swaroop', book='A Byte of Python'))
```

Output:

```
0.333
___hello___
Swaroop wrote A Byte of Python
```

Since we are discussing formatting, note that `print` always ends with an invisible "new line" character (`\n`) so that repeated calls to `print` will all print on a separate line each. To prevent this newline character from being printed, you can specify that it should `end` with a blank:

```
print('a', end='')
print('b', end='')
```

Output is:

```
ab
```

Or you can `end` with a space:

```
print('a', end=' ')
print('b', end=' ')
print('c')
```

Output is:

```
a b c
```

Escape Sequences

Suppose, you want to have a string which contains a single quote (`'`), how will you specify this string? For example, the string is `"What's your name?"` . You cannot specify `'What's your name?'` because Python will be confused as to where the string starts and ends. So, you will have to specify that this single quote does not indicate the end of the string. This can be done with the help of what is called an *escape sequence*. You specify the single quote as `\'` : notice the backslash. Now, you can specify the string as `'What\'s your name?'` .

Another way of specifying this specific string would be `"What's your name?"` i.e. using double quotes. Similarly, you have to use an escape sequence for using a double quote itself in a double quoted string. Also, you have to indicate the backslash itself using the escape sequence `\\` .

What if you wanted to specify a two-line string? One way is to use a triple-quoted string as shown previously or you can use an escape sequence for the newline character - `\n` to indicate the start of a new line. An example is:

```
'This is the first line\nThis is the second line'
```

Another useful escape sequence to know is the tab: `\t`. There are many more escape sequences but I have mentioned only the most useful ones here.

One thing to note is that in a string, a single backslash at the end of the line indicates that the string is continued in the next line, but no newline is added. For example:

```
"This is the first sentence. \
This is the second sentence."
```

is equivalent to

```
"This is the first sentence. This is the second sentence."
```

Raw String

If you need to specify some strings where no special processing such as escape sequences are handled, then what you need is to specify a *raw* string by prefixing `r` or `R` to the string. An example is:

```
r"Newlines are indicated by \n"
```

> **Note for Regular Expression Users**
>
> Always use raw strings when dealing with regular expressions. Otherwise, a lot of backwhacking may be required. For example, backreferences can be referred to as `'\\1'` or `r'\1'`.

Variable

Using just literal constants can soon become boring - we need some way of storing any information and manipulate them as well. This is where *variables* come into the picture. Variables are exactly what the name implies - their value can vary, i.e., you can store anything using a variable. Variables are just parts of your computer's memory where you store some information. Unlike literal constants, you need some method of accessing these variables and hence you give them names.

Identifier Naming

Variables are examples of identifiers. *Identifiers* are names given to identify *something*. There are some rules you have to follow for naming identifiers:

- The first character of the identifier must be a letter of the alphabet (uppercase ASCII or lowercase ASCII or Unicode character) or an underscore (`_`).
- The rest of the identifier name can consist of letters (uppercase ASCII or lowercase ASCII or Unicode character), underscores (`_`) or digits (0-9).
- Identifier names are case-sensitive. For example, `myname` and `myName` are *not* the same. Note the lowercase `n` in the former and the uppercase `N` in the latter.
- Examples of *valid* identifier names are `i` , `name_2_3` . Examples of *invalid* identifier names are `2things` , `this is spaced out` , `my-name` and `>a1b2_c3` .

Data Types

Variables can hold values of different types called *data types*. The basic types are numbers and strings, which we have already discussed. In later chapters, we will see how to create our own types using classes.

Object

Remember, Python refers to anything used in a program as an *object*. This is meant in the generic sense. Instead of saying "the *something*"', we say "the *object*".

> **Note for Object Oriented Programming users**:
>
> Python is strongly object-oriented in the sense that everything is an object including numbers, strings and functions.

We will now see how to use variables along with literal constants. Save the following example and run the program.

How to write Python programs

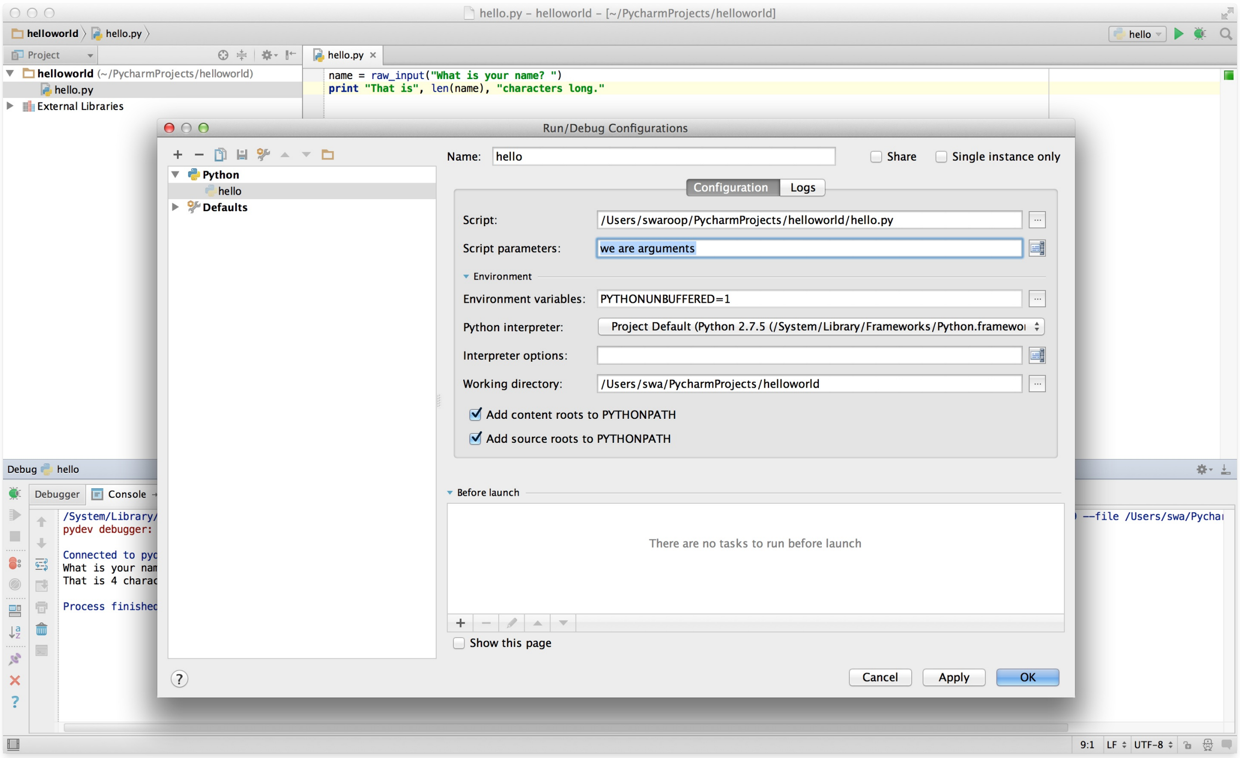
Henceforth, the standard procedure to save and run a Python program is as follows:

For PyCharm

1. Open PyCharm.
2. Create new file with the filename mentioned.
3. Type the program code given in the example.

4. Right-click and run the current file.

NOTE: Whenever you have to provide command line arguments, click on `Run` -> `Edit Configurations` and type the arguments in the `Script parameters:` section and click the `OK` button:



For other editors

1. Open your editor of choice.
2. Type the program code given in the example.
3. Save it as a file with the filename mentioned.
4. Run the interpreter with the command `python program.py` to run the program.

Example: Using Variables And Literal Constants

Type and run the following program:

```
# Filename : var.py
i = 5
print(i)
i = i + 1
print(i)

s = '''This is a multi-line string.
This is the second line.'''
print(s)
```

Output:

```
5
6
This is a multi-line string.
This is the second line.
```

How It Works

Here's how this program works. First, we assign the literal constant value `5` to the variable `i` using the assignment operator (`=`). This line is called a statement because it states that something should be done and in this case, we connect the variable name `i` to the value `5` . Next, we print the value of `i` using the `print` statement which, unsurprisingly, just prints the value of the variable to the screen.

Then we add `1` to the value stored in `i` and store it back. We then print it and expectedly, we get the value `6` .

Similarly, we assign the literal string to the variable `s` and then print it.

> **Note for static language programmers**
>
> Variables are used by just assigning them a value. No declaration or data type definition is needed/used.

Logical And Physical Line

A physical line is what you *see* when you write the program. A logical line is what *Python sees* as a single statement. Python implicitly assumes that each *physical line* corresponds to a *logical line*.

An example of a logical line is a statement like `print 'hello world'` - if this was on a line by itself (as you see it in an editor), then this also corresponds to a physical line.

Implicitly, Python encourages the use of a single statement per line which makes code more readable.

If you want to specify more than one logical line on a single physical line, then you have to explicitly specify this using a semicolon (`;`) which indicates the end of a logical line/statement. For example:

```
i = 5
print(i)
```

is effectively same as

```
i = 5;
print(i);
```

which is also same as

```
i = 5; print(i);
```

and same as

```
i = 5; print(i)
```

However, I *strongly recommend* that you stick to *writing a maximum of a single logical line on each single physical line*. The idea is that you should never use the semicolon. In fact, I have *never* used or even seen a semicolon in a Python program.

There is one kind of situation where this concept is really useful: if you have a long line of code, you can break it into multiple physical lines by using the backslash. This is referred to as *explicit line joining*:

```
s = 'This is a string. \
This continues the string.'
print(s)
```

Output:

```
This is a string. This continues the string.
```

Similarly,

```
i = \
5
```

is the same as

```
i = 5
```

Sometimes, there is an implicit assumption where you don't need to use a backslash. This is the case where the logical line has a starting parentheses, starting square brackets or a starting curly braces but not an ending one. This is called *implicit line joining*. You can see

this in action when we write programs using list in later chapters.

Indentation

Whitespace is important in Python. Actually, *whitespace at the beginning of the line is important*. This is called *indentation*. Leading whitespace (spaces and tabs) at the beginning of the logical line is used to determine the indentation level of the logical line, which in turn is used to determine the grouping of statements.

This means that statements which go together *must* have the same indentation. Each such set of statements is called a *block*. We will see examples of how blocks are important in later chapters.

One thing you should remember is that wrong indentation can give rise to errors. For example:

```
i = 5
# Error below! Notice a single space at the start of the line
 print('Value is', i)
print('I repeat, the value is', i)
```

When you run this, you get the following error:

```
  File "whitespace.py", line 3
    print('Value is', i)
    ^
IndentationError: unexpected indent
```

Notice that there is a single space at the beginning of the second line. The error indicated by Python tells us that the syntax of the program is invalid i.e. the program was not properly written. What this means to you is that *you cannot arbitrarily start new blocks of statements* (except for the default main block which you have been using all along, of course). Cases where you can use new blocks will be detailed in later chapters such as the control flow.

> **How to indent**
>
> Use four spaces for indentation. This is the official Python language recommendation. Good editors will automatically do this for you. Make sure you use a consistent number of spaces for indentation, otherwise your program will not run or will have unexpected behavior.

> **Note to static language programmers**
>
> Python will always use indentation for blocks and will never use braces. Run `from __future__ import braces` to learn more.

Summary

Now that we have gone through many nitty-gritty details, we can move on to more interesting stuff such as control flow statements. Be sure to become comfortable with what you have read in this chapter.

Operators and Expressions

Most statements (logical lines) that you write will contain *expressions*. A simple example of an expression is `2 + 3`. An expression can be broken down into operators and operands.

Operators are functionality that do something and can be represented by symbols such as `+` or by special keywords. Operators require some data to operate on and such data is called *operands*. In this case, `2` and `3` are the operands.

Operators

We will briefly take a look at the operators and their usage.

Note that you can evaluate the expressions given in the examples using the interpreter interactively. For example, to test the expression `2 + 3`, use the interactive Python interpreter prompt:

```
>>> 2 + 3
5
>>> 3 * 5
15
>>>
```

Here is a quick overview of the available operators:

- `+` (plus)
    - Adds two objects
    - `3 + 5` gives `8`. `'a' + 'b'` gives `'ab'`.
- `-` (minus)
    - Gives the subtraction of one number from the other; if the first operand is absent it is assumed to be zero.
    - `-5.2` gives a negative number and `50 - 24` gives `26`.
- `*` (multiply)
    - Gives the multiplication of the two numbers or returns the string repeated that many times.
    - `2 * 3` gives `6`. `'la' * 3` gives `'lalala'`.
- `**` (power)
    - Returns x to the power of y

    - `3 ** 4` gives `81` (i.e. `3 * 3 * 3 * 3`)
- `/` (divide)
    - Divide x by y
    - `13 / 3` gives `4.333333333333333`
- `//` (divide and floor)
    - Divide x by y and round the answer *down* to the nearest integer value. Note that if one of the values is a float, you'll get back a float.
    - `13 // 3` gives `4`
    - `-13 // 3` gives `-5`
    - `9//1.81` gives `4.0`
- `%` (modulo)
    - Returns the remainder of the division
    - `13 % 3` gives `1` . `-25.5 % 2.25` gives `1.5` .
- `<<` (left shift)
    - Shifts the bits of the number to the left by the number of bits specified. (Each number is represented in memory by bits or binary digits i.e. 0 and 1)
    - `2 << 2` gives `8` . `2` is represented by `10` in bits.
    - Left shifting by 2 bits gives `1000` which represents the decimal `8` .
- `>>` (right shift)
    - Shifts the bits of the number to the right by the number of bits specified.
    - `11 >> 1` gives `5` .
    - `11` is represented in bits by `1011` which when right shifted by 1 bit gives `101` which is the decimal `5` .
- `&` (bit-wise AND)
    - Bit-wise AND of the numbers
    - `5 & 3` gives `1` .
- `|` (bit-wise OR)
    - Bitwise OR of the numbers
    - `5 | 3` gives `7`
- `^` (bit-wise XOR)
    - Bitwise XOR of the numbers
    - `5 ^ 3` gives `6`
- `~` (bit-wise invert)
    - The bit-wise inversion of x is -(x+1)
    - `~5` gives `-6` . More details at http://stackoverflow.com/a/11810203

- `<` (less than)
    - Returns whether x is less than y. All comparison operators return `True` or `False`. Note the capitalization of these names.
    - `5 < 3` gives `False` and `3 < 5` gives `True`.
    - Comparisons can be chained arbitrarily: `3 < 5 < 7` gives `True`.
- `>` (greater than)
    - Returns whether x is greater than y
    - `5 > 3` returns `True`. If both operands are numbers, they are first converted to a common type. Otherwise, it always returns `False`.
- `<=` (less than or equal to)
    - Returns whether x is less than or equal to y
    - `x = 3; y = 6; x <= y` returns `True`
- `>=` (greater than or equal to)
    - Returns whether x is greater than or equal to y
    - `x = 4; y = 3; x >= 3` returns `True`
- `==` (equal to)
    - Compares if the objects are equal
    - `x = 2; y = 2; x == y` returns `True`
    - `x = 'str'; y = 'stR'; x == y` returns `False`
    - `x = 'str'; y = 'str'; x == y` returns `True`
- `!=` (not equal to)
    - Compares if the objects are not equal
    - `x = 2; y = 3; x != y` returns `True`
- `not` (boolean NOT)
    - If x is `True`, it returns `False`. If x is `False`, it returns `True`.
    - `x = True; not x` returns `False`.
- `and` (boolean AND)
    - `x and y` returns `False` if x is `False`, else it returns evaluation of y
    - `x = False; y = True; x and y` returns `False` since x is False. In this case, Python will not evaluate y since it knows that the left hand side of the 'and' expression is `False` which implies that the whole expression will be `False` irrespective of the other values. This is called short-circuit evaluation.
- `or` (boolean OR)
    - If x is `True`, it returns True, else it returns evaluation of y
    - `x = True; y = False; x or y` returns `True`. Short-circuit evaluation applies here

as well.

Shortcut for math operation and assignment

It is common to run a math operation on a variable and then assign the result of the operation back to the variable, hence there is a shortcut for such expressions:

```
a = 2
a = a * 3
```

can be written as:

```
a = 2
a *= 3
```

Notice that `var = var operation expression` becomes `var operation= expression`.

Evaluation Order

If you had an expression such as `2 + 3 * 4`, is the addition done first or the multiplication? Our high school maths tells us that the multiplication should be done first. This means that the multiplication operator has higher precedence than the addition operator.

The following table gives the precedence table for Python, from the lowest precedence (least binding) to the highest precedence (most binding). This means that in a given expression, Python will first evaluate the operators and expressions lower in the table before the ones listed higher in the table.

The following table, taken from the Python reference manual, is provided for the sake of completeness. It is far better to use parentheses to group operators and operands appropriately in order to explicitly specify the precedence. This makes the program more readable. See Changing the Order of Evaluation below for details.

- `lambda` : Lambda Expression
- `if - else` : Conditional expression
- `or` : Boolean OR
- `and` : Boolean AND
- `not x` : Boolean NOT
- `in, not in, is, is not, <, <=, >, >=, !=, ==` : Comparisons, including membership tests and identity tests

- `|` : Bitwise OR
- `^` : Bitwise XOR
- `&` : Bitwise AND
- `<<, >>` : Shifts
- `+, -` : Addition and subtraction
- `*, /, //, %` : Multiplication, Division, Floor Division and Remainder
- `+x, -x, ~x` : Positive, Negative, bitwise NOT
- `**` : Exponentiation
- `x[index], x[index:index], x(arguments...), x.attribute` : Subscription, slicing, call, attribute reference
- `(expressions...), [expressions...], {key: value...}, {expressions...}` : Binding or tuple display, list display, dictionary display, set display

The operators which we have not already come across will be explained in later chapters.

Operators with the *same precedence* are listed in the same row in the above table. For example, `+` and `-` have the same precedence.

Changing the Order Of Evaluation

To make the expressions more readable, we can use parentheses. For example, `2 + (3 * 4)` is definitely easier to understand than `2 + 3 * 4` which requires knowledge of the operator precedences. As with everything else, the parentheses should be used reasonably (do not overdo it) and should not be redundant, as in `(2 + (3 * 4))`.

There is an additional advantage to using parentheses - it helps us to change the order of evaluation. For example, if you want addition to be evaluated before multiplication in an expression, then you can write something like `(2 + 3) * 4`.

Associativity

Operators are usually associated from left to right. This means that operators with the same precedence are evaluated in a left to right manner. For example, `2 + 3 + 4` is evaluated as `(2 + 3) + 4`.

Expressions

Example (save as `expression.py`):

```
length = 5
breadth = 2

area = length * breadth
print('Area is', area)
print('Perimeter is', 2 * (length + breadth))
```

Output:

```
$ python expression.py
Area is 10
Perimeter is 14
```

How It Works

The length and breadth of the rectangle are stored in variables by the same name. We use these to calculate the area and perimeter of the rectangle with the help of expressions. We store the result of the expression `length * breadth` in the variable `area` and then print it using the `print` function. In the second case, we directly use the value of the expression `2 * (length + breadth)` in the print function.

Also, notice how Python *pretty-prints* the output. Even though we have not specified a space between `'Area is'` and the variable `area`, Python puts it for us so that we get a clean nice output and the program is much more readable this way (since we don't need to worry about spacing in the strings we use for output). This is an example of how Python makes life easy for the programmer.

Summary

We have seen how to use operators, operands and expressions - these are the basic building blocks of any program. Next, we will see how to make use of these in our programs using statements.

Control Flow

In the programs we have seen till now, there has always been a series of statements faithfully executed by Python in exact top-down order. What if you wanted to change the flow of how it works? For example, you want the program to take some decisions and do different things depending on different situations, such as printing 'Good Morning' or 'Good Evening' depending on the time of the day?

As you might have guessed, this is achieved using control flow statements. There are three control flow statements in Python - `if` , `for` and `while` .

The `if` statement

The `if` statement is used to check a condition: *if* the condition is true, we run a block of statements (called the *if-block*), *else* we process another block of statements (called the *else-block*). The *else* clause is optional.

Example (save as `if.py`):

```
number = 23
guess = int(input('Enter an integer : '))

if guess == number:
    # New block starts here
    print('Congratulations, you guessed it.')
    print('(but you do not win any prizes!)')
    # New block ends here
elif guess < number:
    # Another block
    print('No, it is a little higher than that')
    # You can do whatever you want in a block ...
else:
    print('No, it is a little lower than that')
    # you must have guessed > number to reach here

print('Done')
# This last statement is always executed,
# after the if statement is executed.
```

Output:

```
$ python if.py
Enter an integer : 50
No, it is a little lower than that
Done

$ python if.py
Enter an integer : 22
No, it is a little higher than that
Done

$ python if.py
Enter an integer : 23
Congratulations, you guessed it.
(but you do not win any prizes!)
Done
```

How It Works

In this program, we take guesses from the user and check if it is the number that we have. We set the variable `number` to any integer we want, say `23`. Then, we take the user's guess using the `input()` function. Functions are just reusable pieces of programs. We'll read more about them in the next chapter.

We supply a string to the built-in `input` function which prints it to the screen and waits for input from the user. Once we enter something and press kbd:[enter] key, the `input()` function returns what we entered, as a string. We then convert this string to an integer using `int` and then store it in the variable `guess`. Actually, the `int` is a class but all you need to know right now is that you can use it to convert a string to an integer (assuming the string contains a valid integer in the text).

Next, we compare the guess of the user with the number we have chosen. If they are equal, we print a success message. Notice that we use indentation levels to tell Python which statements belong to which block. This is why indentation is so important in Python. I hope you are sticking to the "consistent indentation" rule. Are you?

Notice how the `if` statement contains a colon at the end - we are indicating to Python that a block of statements follows.

Then, we check if the guess is less than the number, and if so, we inform the user that they must guess a little higher than that. What we have used here is the `elif` clause which actually combines two related `if else-if else` statements into one combined `if-elif-else` statement. This makes the program easier and reduces the amount of indentation required.

The `elif` and `else` statements must also have a colon at the end of the logical line followed by their corresponding block of statements (with proper indentation, of course)

You can have another `if` statement inside the if-block of an `if` statement and so on - this is called a nested `if` statement.

Remember that the `elif` and `else` parts are optional. A minimal valid `if` statement is:

```
if True:
    print('Yes, it is true')
```

After Python has finished executing the complete `if` statement along with the associated `elif` and `else` clauses, it moves on to the next statement in the block containing the `if` statement. In this case, it is the main block (where execution of the program starts), and the next statement is the `print('Done')` statement. After this, Python sees the ends of the program and simply finishes up.

Even though this is a very simple program, I have been pointing out a lot of things that you should notice. All these are pretty straightforward (and surprisingly simple for those of you from C/C++ backgrounds). You will need to become aware of all these things initially, but after some practice you will become comfortable with them, and it will all feel 'natural' to you.

> **Note for C/C++ Programmers**
>
> There is no `switch` statement in Python. You can use an `if..elif..else` statement to do the same thing (and in some cases, use a dictionary to do it quickly)

The while Statement

The `while` statement allows you to repeatedly execute a block of statements as long as a condition is true. A `while` statement is an example of what is called a *looping* statement. A `while` statement can have an optional `else` clause.

Example (save as `while.py`):

```
number = 23
running = True

while running:
    guess = int(input('Enter an integer : '))

    if guess == number:
        print('Congratulations, you guessed it.')
        # this causes the while loop to stop
        running = False
    elif guess < number:
        print('No, it is a little higher than that.')
    else:
        print('No, it is a little lower than that.')
else:
    print('The while loop is over.')
    # Do anything else you want to do here

print('Done')
```

Output:

```
$ python while.py
Enter an integer : 50
No, it is a little lower than that.
Enter an integer : 22
No, it is a little higher than that.
Enter an integer : 23
Congratulations, you guessed it.
The while loop is over.
Done
```

How It Works

In this program, we are still playing the guessing game, but the advantage is that the user is allowed to keep guessing until he guesses correctly - there is no need to repeatedly run the program for each guess, as we have done in the previous section. This aptly demonstrates the use of the `while` statement.

We move the `input` and `if` statements to inside the `while` loop and set the variable `running` to `True` before the while loop. First, we check if the variable `running` is `True` and then proceed to execute the corresponding *while-block*. After this block is executed, the condition is again checked which in this case is the `running` variable. If it is true, we execute the while-block again, else we continue to execute the optional else-block and then continue to the next statement.

The `else` block is executed when the `while` loop condition becomes `False` - this may even be the first time that the condition is checked. If there is an `else` clause for a `while` loop, it is always executed unless you break out of the loop with a `break` statement.

The `True` and `False` are called Boolean types and you can consider them to be equivalent to the value `1` and `0` respectively.

> **Note for C/C++ Programmers**
>
> Remember that you can have an `else` clause for the `while` loop.

The `for` loop

The `for..in` statement is another looping statement which *iterates* over a sequence of objects i.e. go through each item in a sequence. We will see more about sequences in detail in later chapters. What you need to know right now is that a sequence is just an ordered collection of items.

Example (save as `for.py`):

```
for i in range(1, 5):
    print(i)
else:
    print('The for loop is over')
```

Output:

```
$ python for.py
1
2
3
4
The for loop is over
```

How It Works

In this program, we are printing a *sequence* of numbers. We generate this sequence of numbers using the built-in `range` function.

What we do here is supply it two numbers and `range` returns a sequence of numbers starting from the first number and up to the second number. For example, `range(1,5)` gives the sequence `[1, 2, 3, 4]`. By default, `range` takes a step count of 1. If we supply a third

number to `range` , then that becomes the step count. For example, `range(1,5,2)` gives `[1,3]` . Remember that the range extends *up to* the second number i.e. it does *not* include the second number.

Note that `range()` generates only one number at a time, if you want the full list of numbers, call `list()` on the `range()` , for example, `list(range(5))` will result in `[0, 1, 2, 3, 4]` . Lists are explained in the data structures chapter.

The `for` loop then iterates over this range - `for i in range(1,5)` is equivalent to `for i in [1, 2, 3, 4]` which is like assigning each number (or object) in the sequence to i, one at a time, and then executing the block of statements for each value of `i` . In this case, we just print the value in the block of statements.

Remember that the `else` part is optional. When included, it is always executed once after the `for` loop is over unless a break statement is encountered.

Remember that the `for..in` loop works for any sequence. Here, we have a list of numbers generated by the built-in `range` function, but in general we can use any kind of sequence of any kind of objects! We will explore this idea in detail in later chapters.

> **Note for C/C++/Java/C# Programmers**
>
> The Python `for` loop is radically different from the C/C++ `for` loop. C# programmers will note that the `for` loop in Python is similar to the `foreach` loop in C#. Java programmers will note that the same is similar to `for (int i : IntArray)` in Java 1.5.
>
> In C/C++, if you want to write `for (int i = 0; i < 5; i++)` , then in Python you write just `for i in range(0,5)` . As you can see, the `for` loop is simpler, more expressive and less error prone in Python.

The break Statement

The `break` statement is used to *break* out of a loop statement i.e. stop the execution of a looping statement, even if the loop condition has not become `False` or the sequence of items has not been completely iterated over.

An important note is that if you *break* out of a `for` or `while` loop, any corresponding loop `else` block is **not** executed.

Example (save as `break.py`):

```python
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    print('Length of the string is', len(s))
print('Done')
```

Output:

```
$ python break.py
Enter something : Programming is fun
Length of the string is 18
Enter something : When the work is done
Length of the string is 21
Enter something : if you wanna make your work also fun:
Length of the string is 37
Enter something : use Python!
Length of the string is 11
Enter something : quit
Done
```

How It Works

In this program, we repeatedly take the user's input and print the length of each input each time. We are providing a special condition to stop the program by checking if the user input is `'quit'`. We stop the program by *breaking* out of the loop and reach the end of the program.

The length of the input string can be found out using the built-in `len` function.

Remember that the `break` statement can be used with the `for` loop as well.

Swaroop's Poetic Python

The input I have used here is a mini poem I have written:

```
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```

The `continue` Statement

The `continue` statement is used to tell Python to skip the rest of the statements in the current loop block and to *continue* to the next iteration of the loop.

Example (save as `continue.py`):

```
while True:
    s = input('Enter something : ')
    if s == 'quit':
        break
    if len(s) < 3:
        print('Too small')
        continue
    print('Input is of sufficient length')
    # Do other kinds of processing here...
```

Output:

```
$ python continue.py
Enter something : a
Too small
Enter something : 12
Too small
Enter something : abc
Input is of sufficient length
Enter something : quit
```

How It Works

In this program, we accept input from the user, but we process the input string only if it is at least 3 characters long. So, we use the built-in `len` function to get the length and if the length is less than 3, we skip the rest of the statements in the block by using the `continue` statement. Otherwise, the rest of the statements in the loop are executed, doing any kind of processing we want to do here.

Note that the `continue` statement works with the `for` loop as well.

Summary

We have seen how to use the three control flow statements - `if`, `while` and `for` along with their associated `break` and `continue` statements. These are some of the most commonly used parts of Python and hence, becoming comfortable with them is essential.

Next, we will see how to create and use functions.

Functions

Functions are reusable pieces of programs. They allow you to give a name to a block of statements, allowing you to run that block using the specified name anywhere in your program and any number of times. This is known as *calling* the function. We have already used many built-in functions such as `len` and `range`.

The function concept is probably *the* most important building block of any non-trivial software (in any programming language), so we will explore various aspects of functions in this chapter.

Functions are defined using the `def` keyword. After this keyword comes an *identifier* name for the function, followed by a pair of parentheses which may enclose some names of variables, and by the final colon that ends the line. Next follows the block of statements that are part of this function. An example will show that this is actually very simple:

Example (save as `function1.py`):

```
def say_hello():
    # block belonging to the function
    print('hello world')
# End of function

say_hello()  # call the function
say_hello()  # call the function again
```

Output:

```
$ python function1.py
hello world
hello world
```

How It Works

We define a function called `say_hello` using the syntax as explained above. This function takes no parameters and hence there are no variables declared in the parentheses. Parameters to functions are just input to the function so that we can pass in different values to it and get back corresponding results.

Notice that we can call the same function twice which means we do not have to write the same code again.

Function Parameters

A function can take parameters, which are values you supply to the function so that the function can *do* something utilising those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are already assigned values when the function runs.

Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. Note the terminology used - the names given in the function definition are called *parameters* whereas the values you supply in the function call are called *arguments*.

Example (save as `function_param.py`):

```
def print_max(a, b):
    if a > b:
        print(a, 'is maximum')
    elif a == b:
        print(a, 'is equal to', b)
    else:
        print(b, 'is maximum')

# directly pass literal values
print_max(3, 4)

x = 5
y = 7

# pass variables as arguments
print_max(x, y)
```

Output:

```
$ python function_param.py
4 is maximum
7 is maximum
```

How It Works

Here, we define a function called `print_max` that uses two parameters called `a` and `b` . We find out the greater number using a simple `if..else` statement and then print the bigger number.

The first time we call the function `print_max` , we directly supply the numbers as arguments. In the second case, we call the function with variables as arguments. `print_max(x, y)` causes the value of argument `x` to be assigned to parameter `a` and the value of

argument `y` to be assigned to parameter `b`. The `print_max` function works the same way in both cases.

Local Variables

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function - i.e. variable names are *local* to the function. This is called the *scope* of the variable. All variables have the scope of the block they are declared in starting from the point of definition of the name.

Example (save as `function_local.py`):

```
x = 50


def func(x):
    print('x is', x)
    x = 2
    print('Changed local x to', x)


func(x)
print('x is still', x)
```

Output:

```
$ python function_local.py
x is 50
Changed local x to 2
x is still 50
```

How It Works

The first time that we print the *value* of the name *x* with the first line in the function's body, Python uses the value of the parameter declared in the main block, above the function definition.

Next, we assign the value `2` to `x`. The name `x` is local to our function. So, when we change the value of `x` in the function, the `x` defined in the main block remains unaffected.

With the last `print` statement, we display the value of `x` as defined in the main block, thereby confirming that it is actually unaffected by the local assignment within the previously called function.

The `global` statement

If you want to assign a value to a name defined at the top level of the program (i.e. not inside any kind of scope such as functions or classes), then you have to tell Python that the name is not local, but it is *global*. We do this using the `global` statement. It is impossible to assign a value to a variable defined outside a function without the `global` statement.

You can use the values of such variables defined outside the function (assuming there is no variable with the same name within the function). However, this is not encouraged and should be avoided since it becomes unclear to the reader of the program as to where that variable's definition is. Using the `global` statement makes it amply clear that the variable is defined in an outermost block.

Example (save as `function_global.py`):

```
x = 50


def func():
    global x

    print('x is', x)
    x = 2
    print('Changed global x to', x)


func()
print('Value of x is', x)
```

Output:

```
$ python function_global.py
x is 50
Changed global x to 2
Value of x is 2
```

How It Works

The `global` statement is used to declare that `x` is a global variable - hence, when we assign a value to `x` inside the function, that change is reflected when we use the value of `x` in the main block.

You can specify more than one global variable using the same `global` statement e.g. `global x, y, z`.

Default Argument Values

For some functions, you may want to make some parameters *optional* and use default values in case the user does not want to provide values for them. This is done with the help of default argument values. You can specify default argument values for parameters by appending to the parameter name in the function definition the assignment operator (`=`) followed by the default value.

Note that the default argument value should be a constant. More precisely, the default argument value should be immutable - this is explained in detail in later chapters. For now, just remember this.

Example (save as `function_default.py`):

```
def say(message, times=1):
    print(message * times)

say('Hello')
say('World', 5)
```

Output:

```
$ python function_default.py
Hello
WorldWorldWorldWorldWorld
```

How It Works

The function named `say` is used to print a string as many times as specified. If we don't supply a value, then by default, the string is printed just once. We achieve this by specifying a default argument value of `1` to the parameter `times` .

In the first usage of `say` , we supply only the string and it prints the string once. In the second usage of `say` , we supply both the string and an argument `5` stating that we want to *say* the string message 5 times.

> *CAUTION*
>
> Only those parameters which are at the end of the parameter list can be given default argument values i.e. you cannot have a parameter with a default argument value preceding a parameter without a default argument value in the function's parameter list.
>
> This is because the values are assigned to the parameters by position. For example, `def func(a, b=5)` is valid, but `def func(a=5, b)` is *not valid*.

Keyword Arguments

If you have some functions with many parameters and you want to specify only some of them, then you can give values for such parameters by naming them - this is called *keyword arguments* - we use the name (keyword) instead of the position (which we have been using all along) to specify the arguments to the function.

There are two advantages - one, using the function is easier since we do not need to worry about the order of the arguments. Two, we can give values to only those parameters to which we want to, provided that the other parameters have default argument values.

Example (save as `function_keyword.py`):

```
def func(a, b=5, c=10):
    print('a is', a, 'and b is', b, 'and c is', c)

func(3, 7)
func(25, c=24)
func(c=50, a=100)
```

Output:

```
$ python function_keyword.py
a is 3 and b is 7 and c is 10
a is 25 and b is 5 and c is 24
a is 100 and b is 5 and c is 50
```

How It Works

The function named `func` has one parameter without a default argument value, followed by two parameters with default argument values.

In the first usage, `func(3, 7)`, the parameter `a` gets the value `3`, the parameter `b` gets the value `7` and `c` gets the default value of `10`.

In the second usage `func(25, c=24)`, the variable `a` gets the value of 25 due to the position of the argument. Then, the parameter `c` gets the value of `24` due to naming i.e. keyword arguments. The variable `b` gets the default value of `5`.

In the third usage `func(c=50, a=100)`, we use keyword arguments for all specified values. Notice that we are specifying the value for parameter `c` before that for `a` even though `a` is defined before `c` in the function definition.

VarArgs parameters

Sometimes you might want to define a function that can take *any* number of parameters, i.e. **var**iable number of **arg**uments, this can be achieved by using the stars (save as `function_varargs.py`):

```
def total(a=5, *numbers, **phonebook):
    print('a', a)

    #iterate through all the items in tuple
    for single_item in numbers:
        print('single_item', single_item)

    #iterate through all the items in dictionary
    for first_part, second_part in phonebook.items():
        print(first_part,second_part)

print(total(10,1,2,3,Jack=1123,John=2231,Inge=1560))
```

Output:

```
$ python function_varargs.py
a 10
single_item 1
single_item 2
single_item 3
Inge 1560
John 2231
Jack 1123
None
```

How It Works

When we declare a starred parameter such as `*param`, then all the positional arguments from that point till the end are collected as a tuple called 'param'.

Similarly, when we declare a double-starred parameter such as `**param`, then all the keyword arguments from that point till the end are collected as a dictionary called 'param'.

We will explore tuples and dictionaries in a later chapter.

The `return` statement

The `return` statement is used to *return* from a function i.e. break out of the function. We can optionally *return a value* from the function as well.

Example (save as `function_return.py`):

```python
def maximum(x, y):
    if x > y:
        return x
    elif x == y:
        return 'The numbers are equal'
    else:
        return y

print(maximum(2, 3))
```

Output:

```
$ python function_return.py
3
```

How It Works

The `maximum` function returns the maximum of the parameters, in this case the numbers supplied to the function. It uses a simple `if..else` statement to find the greater value and then *returns* that value.

Note that a `return` statement without a value is equivalent to `return None`. `None` is a special type in Python that represents nothingness. For example, it is used to indicate that a variable has no value if it has a value of `None`.

Every function implicitly contains a `return None` statement at the end unless you have written your own `return` statement. You can see this by running `print(some_function())` where the function `some_function` does not use the `return` statement such as:

```python
def some_function():
    pass
```

The `pass` statement is used in Python to indicate an empty block of statements.

> TIP: There is a built-in function called `max` that already implements the 'find maximum' functionality, so use this built-in function whenever possible.

DocStrings

Python has a nifty feature called *documentation strings*, usually referred to by its shorter name *docstrings*. DocStrings are an important tool that you should make use of since it helps to document the program better and makes it easier to understand. Amazingly, we can even get the docstring back from, say a function, when the program is actually running!

Example (save as `function_docstring.py`):

```
def print_max(x, y):
    '''Prints the maximum of two numbers.

    The two values must be integers.'''
    # convert to integers, if possible
    x = int(x)
    y = int(y)

    if x > y:
        print(x, 'is maximum')
    else:
        print(y, 'is maximum')

print_max(3, 5)
print(print_max.__doc__)
```

Output:

```
$ python function_docstring.py
5 is maximum
Prints the maximum of two numbers.

    The two values must be integers.
```

How It Works

A string on the first logical line of a function is the *docstring* for that function. Note that DocStrings also apply to modules and classes which we will learn about in the respective chapters.

The convention followed for a docstring is a multi-line string where the first line starts with a capital letter and ends with a dot. Then the second line is blank followed by any detailed explanation starting from the third line. You are *strongly advised* to follow this convention for all your docstrings for all your non-trivial functions.

We can access the docstring of the `print_max` function using the `__doc__` (notice the *double underscores*) attribute (name belonging to) of the function. Just remember that Python treats *everything* as an object and this includes functions. We'll learn more about objects in the chapter on classes.

If you have used `help()` in Python, then you have already seen the usage of docstrings! What it does is just fetch the `__doc__` attribute of that function and displays it in a neat manner for you. You can try it out on the function above - just include `help(print_max)` in your program. Remember to press the `q` key to exit `help`.

Automated tools can retrieve the documentation from your program in this manner. Therefore, I *strongly recommend* that you use docstrings for any non-trivial function that you write. The `pydoc` command that comes with your Python distribution works similarly to `help()` using docstrings.

Summary

We have seen so many aspects of functions but note that we still haven't covered all aspects of them. However, we have already covered most of what you'll use regarding Python functions on an everyday basis.

Next, we will see how to use as well as create Python modules.

Modules

You have seen how you can reuse code in your program by defining functions once. What if you wanted to reuse a number of functions in other programs that you write? As you might have guessed, the answer is modules.

There are various methods of writing modules, but the simplest way is to create a file with a `.py` extension that contains functions and variables.

Another method is to write the modules in the native language in which the Python interpreter itself was written. For example, you can write modules in the C programming language and when compiled, they can be used from your Python code when using the standard Python interpreter.

A module can be *imported* by another program to make use of its functionality. This is how we can use the Python standard library as well. First, we will see how to use the standard library modules.

Example (save as `module_using_sys.py`):

```
import sys

print('The command line arguments are:')
for i in sys.argv:
    print(i)

print('\n\nThe PYTHONPATH is', sys.path, '\n')
```

Output:

```
$ python module_using_sys.py we are arguments    # each arg is separated by white spac
e
The command line arguments are:
module_using_sys.py
we
are
arguments


The PYTHONPATH is ['/tmp/py',
# many entries here, not shown here
'/Library/Python/2.7/site-packages',
'/usr/local/lib/python2.7/site-packages']
```

How It Works

First, we *import* the `sys` module using the `import` statement. Basically, this translates to us telling Python that we want to use this module. The `sys` module contains functionality related to the Python interpreter and its environment i.e. the **sys**tem.

When Python executes the `import sys` statement, it looks for the `sys` module. In this case, it is one of the built-in modules, and hence Python knows where to find it.

If it was not a compiled module i.e. a module written in Python, then the Python interpreter will search for it in the directories listed in its `sys.path` variable. If the module is found, then the statements in the body of that module are run and the module is made *available* for you to use. Note that the initialization is done only the *first* time that we import a module.

The `argv` variable in the `sys` module is accessed using the dotted notation i.e. `sys.argv`. It clearly indicates that this name is part of the `sys` module. Another advantage of this approach is that the name does not clash with any `argv` variable used in your program.

The `sys.argv` variable is a *list* of strings (lists are explained in detail in a later chapter). Specifically, the `sys.argv` contains the list of *command line arguments* i.e. the arguments passed to your program using the command line.

If you are using an IDE to write and run these programs, look for a way to specify command line arguments to the program in the menus.

Here, when we execute `python module_using_sys.py we are arguments`, we run the module `module_using_sys.py` with the `python` command and the other things that follow are arguments passed to the program. Python stores the command line arguments in the `sys.argv` variable for us to use.

Remember, the name of the script running is always the first element in the `sys.argv` list. So, in this case we will have `'module_using_sys.py'` as `sys.argv[0]`, `'we'` as `sys.argv[1]`, `'are'` as `sys.argv[2]` and `'arguments'` as `sys.argv[3]`. Notice that Python starts counting from 0 and not 1.

The `sys.path` contains the list of directory names where modules are imported from. Observe that the first string in `sys.path` is empty - this empty string indicates that the current directory is also part of the `sys.path` which is same as the `PYTHONPATH` environment variable. This means that you can directly import modules located in the current directory. Otherwise, you will have to place your module in one of the directories listed in `sys.path`.

Note that the current directory is the directory from which the program is launched. Run `import os; print(os.getcwd())` to find out the current directory of your program.

Byte-compiled .pyc files

Importing a module is a relatively costly affair, so Python does some tricks to make it faster. One way is to create *byte-compiled* files with the extension `.pyc` which is an intermediate form that Python transforms the program into (remember the introduction section on how Python works?). This `.pyc` file is useful when you import the module the next time from a different program - it will be much faster since a portion of the processing required in importing a module is already done. Also, these byte-compiled files are platform-independent.

NOTE: These `.pyc` files are usually created in the same directory as the corresponding `.py` files. If Python does not have permission to write to files in that directory, then the `.pyc` files will *not* be created.

The from..import statement

If you want to directly import the `argv` variable into your program (to avoid typing the `sys.` everytime for it), then you can use the `from sys import argv` statement.

> WARNING: In general, *avoid* using the `from..import` statement, use the `import` statement instead. This is because your program will avoid name clashes and will be more readable.

Example:

```
from math import sqrt
print("Square root of 16 is", sqrt(16))
```

A module's `__name__`

Every module has a name and statements in a module can find out the name of their module. This is handy for the particular purpose of figuring out whether the module is being run standalone or being imported. As mentioned previously, when a module is imported for the first time, the code it contains gets executed. We can use this to make the module behave in different ways depending on whether it is being used by itself or being imported from another module. This can be achieved using the `__name__` attribute of the module.

Example (save as `module_using_name.py`):

```
if __name__ == '__main__':
    print('This program is being run by itself')
else:
    print('I am being imported from another module')
```

Output:

```
$ python module_using_name.py
This program is being run by itself

$ python
>>> import module_using_name
I am being imported from another module
>>>
```

How It Works

Every Python module has its `__name__` defined. If this is `'__main__'`, that implies that the module is being run standalone by the user and we can take appropriate actions.

Making Your Own Modules

Creating your own modules is easy, you've been doing it all along! This is because every Python program is also a module. You just have to make sure it has a `.py` extension. The following example should make it clear.

Example (save as `mymodule.py`):

```
def say_hi():
    print('Hi, this is mymodule speaking.')

__version__ = '0.1'
```

The above was a sample *module*. As you can see, there is nothing particularly special about it compared to our usual Python program. We will next see how to use this module in our other Python programs.

Remember that the module should be placed either in the same directory as the program from which we import it, or in one of the directories listed in `sys.path`.

Another module (save as `mymodule_demo.py`):

```python
import mymodule

mymodule.say_hi()
print('Version', mymodule.__version__)
```

Output:

```
$ python mymodule_demo.py
Hi, this is mymodule speaking.
Version 0.1
```

How It Works

Notice that we use the same dotted notation to access members of the module. Python makes good reuse of the same notation to give the distinctive 'Pythonic' feel to it so that we don't have to keep learning new ways to do things.

Here is a version utilising the `from..import` syntax (save as `mymodule_demo2.py`):

```python
from mymodule import say_hi, __version__

say_hi()
print('Version', __version__)
```

The output of `mymodule_demo2.py` is same as the output of `mymodule_demo.py`.

Notice that if there was already a `__version__` name declared in the module that imports mymodule, there would be a clash. This is also likely because it is common practice for each module to declare it's version number using this name. Hence, it is always recommended to prefer the `import` statement even though it might make your program a little longer.

You could also use:

```python
from mymodule import *
```

This will import all public names such as `say_hi` but would not import `__version__` because it starts with double underscores.

> WARNING: Remember that you should avoid using import-star, i.e. `from mymodule import *`.

> **Zen of Python**
>
> One of Python's guiding principles is that "Explicit is better than Implicit". Run `import this` in Python to learn more.

The `dir` function

Built-in `dir()` function returns list of names defined by an object. If the object is a module, this list includes functions, classes and variables, defined inside that module.

This function can accept arguments. If the argument is the name of the module, function returns list of names from that specified module. If there is no argument, function returns list of names from the current module.

Example:

```
$ python
>>> import sys

# get names of attributes in sys module
>>> dir(sys)
['__displayhook__', '__doc__',
'argv', 'builtin_module_names',
'version', 'version_info']
# only few entries shown here

# get names of attributes for current module
>>> dir()
['__builtins__', '__doc__',
'__name__', '__package__', 'sys']

# create a new variable 'a'
>>> a = 5

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys', 'a']

# delete/remove a name
>>> del a

>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'sys']
```

How It Works

First, we see the usage of `dir` on the imported `sys` module. We can see the huge list of attributes that it contains.

Next, we use the `dir` function without passing parameters to it. By default, it returns the list of attributes for the current module. Notice that the list of imported modules is also part of this list.

In order to observe the `dir` in action, we define a new variable `a` and assign it a value and then check `dir` and we observe that there is an additional value in the list of the same name. We remove the variable/attribute of the current module using the `del` statement and the change is reflected again in the output of the `dir` function.

A note on `del` - this statement is used to *delete* a variable/name and after the statement has run, in this case `del a`, you can no longer access the variable `a` - it is as if it never existed before at all.

Note that the `dir()` function works on *any* object. For example, run `dir(str)` for the attributes of the `str` (string) class.

There is also a `vars()` function which can potentially give you the attributes and their values, but it will not work for all cases.

Packages

By now, you must have started observing the hierarchy of organizing your programs. Variables usually go inside functions. Functions and global variables usually go inside modules. What if you wanted to organize modules? That's where packages come into the picture.

Packages are just folders of modules with a special `__init__.py` file that indicates to Python that this folder is special because it contains Python modules.

Let's say you want to create a package called 'world' with subpackages 'asia', 'africa', etc. and these subpackages in turn contain modules like 'india', 'madagascar', etc.

This is how you would structure the folders:

```
- <some folder present in the sys.path>/
    - world/
        - __init__.py
        - asia/
            - __init__.py
            - india/
                - __init__.py
                - foo.py
        - africa/
            - __init__.py
            - madagascar/
                - __init__.py
                - bar.py
```

Packages are just a convenience to hierarchically organize modules. You will see many instances of this in the standard library.

Summary

Just like functions are reusable parts of programs, modules are reusable programs. Packages are another hierarchy to organize modules. The standard library that comes with Python is an example of such a set of packages and modules.

We have seen how to use these modules and create our own modules.

Next, we will learn about some interesting concepts called data structures.

Data Structures

Data structures are basically just that - they are *structures* which can hold some *data* together. In other words, they are used to store a collection of related data.

There are four built-in data structures in Python - *list, tuple, dictionary and set*. We will see how to use each of them and how they make life easier for us.

List

A `list` is a data structure that holds an ordered collection of items i.e. you can store a *sequence* of items in a list. This is easy to imagine if you can think of a shopping list where you have a list of items to buy, except that you probably have each item on a separate line in your shopping list whereas in Python you put commas in between them.

The list of items should be enclosed in square brackets so that Python understands that you are specifying a list. Once you have created a list, you can add, remove or search for items in the list. Since we can add and remove items, we say that a list is a *mutable* data type i.e. this type can be altered.

Quick Introduction To Objects And Classes

Although I've been generally delaying the discussion of objects and classes till now, a little explanation is needed right now so that you can understand lists better. We will explore this topic in detail in a later chapter.

A list is an example of usage of objects and classes. When we use a variable `i` and assign a value to it, say integer `5` to it, you can think of it as creating an *object* (i.e. instance) `i` of *class* (i.e. type) `int`. In fact, you can read `help(int)` to understand this better.

A class can also have *methods* i.e. functions defined for use with respect to that class only. You can use these pieces of functionality only when you have an object of that class. For example, Python provides an `append` method for the `list` class which allows you to add an item to the end of the list. For example, `mylist.append('an item')` will add that string to the list `mylist`. Note the use of dotted notation for accessing methods of the objects.

A class can also have *fields* which are nothing but variables defined for use with respect to that class only. You can use these variables/names only when you have an object of that class. Fields are also accessed by the dotted notation, for example, `mylist.field`.

Example (save as `ds_using_list.py`):

```
# This is my shopping list
shoplist = ['apple', 'mango', 'carrot', 'banana']

print('I have', len(shoplist), 'items to purchase.')

print('These items are:', end=' ')
for item in shoplist:
    print(item, end=' ')

print('\nI also have to buy rice.')
shoplist.append('rice')
print('My shopping list is now', shoplist)

print('I will sort my list now')
shoplist.sort()
print('Sorted shopping list is', shoplist)

print('The first item I will buy is', shoplist[0])
olditem = shoplist[0]
del shoplist[0]
print('I bought the', olditem)
print('My shopping list is now', shoplist)
```

Output:

```
$ python ds_using_list.py
I have 4 items to purchase.
These items are: apple mango carrot banana
I also have to buy rice.
My shopping list is now ['apple', 'mango', 'carrot', 'banana', 'rice']
I will sort my list now
Sorted shopping list is ['apple', 'banana', 'carrot', 'mango', 'rice']
The first item I will buy is apple
I bought the apple
My shopping list is now ['banana', 'carrot', 'mango', 'rice']
```

How It Works

The variable `shoplist` is a shopping list for someone who is going to the market. In `shoplist` , we only store strings of the names of the items to buy but you can add *any kind of object* to a list including numbers and even other lists.

We have also used the `for..in` loop to iterate through the items of the list. By now, you must have realised that a list is also a sequence. The speciality of sequences will be discussed in a later section.

Notice the use of the `end` parameter in the call to `print` function to indicate that we want to end the output with a space instead of the usual line break.

Next, we add an item to the list using the `append` method of the list object, as already discussed before. Then, we check that the item has been indeed added to the list by printing the contents of the list by simply passing the list to the `print` function which prints it neatly.

Then, we sort the list by using the `sort` method of the list. It is important to understand that this method affects the list itself and does not return a modified list - this is different from the way strings work. This is what we mean by saying that lists are *mutable* and that strings are *immutable*.

Next, when we finish buying an item in the market, we want to remove it from the list. We achieve this by using the `del` statement. Here, we mention which item of the list we want to remove and the `del` statement removes it from the list for us. We specify that we want to remove the first item from the list and hence we use `del shoplist[0]` (remember that Python starts counting from 0).

If you want to know all the methods defined by the list object, see `help(list)` for details.

Tuple

Tuples are used to hold together multiple objects. Think of them as similar to lists, but without the extensive functionality that the list class gives you. One major feature of tuples is that they are *immutable* like strings i.e. you cannot modify tuples.

Tuples are defined by specifying items separated by commas within an optional pair of parentheses.

Tuples are usually used in cases where a statement or a user-defined function can safely assume that the collection of values (i.e. the tuple of values used) will not change.

Example (save as `ds_using_tuple.py`):

```
# I would recommend always using parentheses
# to indicate start and end of tuple
# even though parentheses are optional.
# Explicit is better than implicit.
zoo = ('python', 'elephant', 'penguin')
print('Number of animals in the zoo is', len(zoo))

new_zoo = 'monkey', 'camel', zoo    # parentheses not required but are a good idea
print('Number of cages in the new zoo is', len(new_zoo))
print('All animals in new zoo are', new_zoo)
print('Animals brought from old zoo are', new_zoo[2])
print('Last animal brought from old zoo is', new_zoo[2][2])
print('Number of animals in the new zoo is',
      len(new_zoo)-1+len(new_zoo[2]))
```

Output:

```
$ python ds_using_tuple.py
Number of animals in the zoo is 3
Number of cages in the new zoo is 3
All animals in new zoo are ('monkey', 'camel', ('python', 'elephant', 'penguin'))
Animals brought from old zoo are ('python', 'elephant', 'penguin')
Last animal brought from old zoo is penguin
Number of animals in the new zoo is 5
```

How It Works

The variable `zoo` refers to a tuple of items. We see that the `len` function can be used to get the length of the tuple. This also indicates that a tuple is a sequence as well.

We are now shifting these animals to a new zoo since the old zoo is being closed. Therefore, the `new_zoo` tuple contains some animals which are already there along with the animals brought over from the old zoo. Back to reality, note that a tuple within a tuple does not lose its identity.

We can access the items in the tuple by specifying the item's position within a pair of square brackets just like we did for lists. This is called the *indexing* operator. We access the third item in `new_zoo` by specifying `new_zoo[2]` and we access the third item within the third item in the `new_zoo` tuple by specifying `new_zoo[2][2]`. This is pretty simple once you've understood the idiom.

> **Tuple with 0 or 1 items**
>
> An empty tuple is constructed by an empty pair of parentheses such as `myempty = ()`. However, a tuple with a single item is not so simple. You have to specify it using a comma following the first (and only) item so that Python can differentiate between a tuple and a pair of parentheses surrounding the object in an expression i.e. you have to specify `singleton = (2 , )` if you mean you want a tuple containing the item `2`.

> **Note for Perl programmers**
>
> A list within a list does not lose its identity i.e. lists are not flattened as in Perl. The same applies to a tuple within a tuple, or a tuple within a list, or a list within a tuple, etc. As far as Python is concerned, they are just objects stored using another object, that's all.

Dictionary

A dictionary is like an address-book where you can find the address or contact details of a person by knowing only his/her name i.e. we associate *keys* (name) with *values* (details). Note that the key must be unique just like you cannot find out the correct information if you have two persons with the exact same name.

Note that you can use only immutable objects (like strings) for the keys of a dictionary but you can use either immutable or mutable objects for the values of the dictionary. This basically translates to say that you should use only simple objects for keys.

Pairs of keys and values are specified in a dictionary by using the notation `d = {key1 : value1, key2 : value2 }`. Notice that the key-value pairs are separated by a colon and the pairs are separated themselves by commas and all this is enclosed in a pair of curly braces.

Remember that key-value pairs in a dictionary are not ordered in any manner. If you want a particular order, then you will have to sort them yourself before using it.

The dictionaries that you will be using are instances/objects of the `dict` class.

Example (save as `ds_using_dict.py`):

```
# 'ab' is short for 'a'ddress'b'ook

ab = {
    'Swaroop': 'swaroop@swaroopch.com',
    'Larry': 'larry@wall.org',
    'Matsumoto': 'matz@ruby-lang.org',
    'Spammer': 'spammer@hotmail.com'
}

print("Swaroop's address is", ab['Swaroop'])

# Deleting a key-value pair
del ab['Spammer']

print('\nThere are {} contacts in the address-book\n'.format(len(ab)))

for name, address in ab.items():
    print('Contact {} at {}'.format(name, address))

# Adding a key-value pair
ab['Guido'] = 'guido@python.org'

if 'Guido' in ab:
    print("\nGuido's address is", ab['Guido'])
```

Output:

```
$ python ds_using_dict.py
Swaroop's address is swaroop@swaroopch.com

There are 3 contacts in the address-book

Contact Swaroop at swaroop@swaroopch.com
Contact Matsumoto at matz@ruby-lang.org
Contact Larry at larry@wall.org

Guido's address is guido@python.org
```

How It Works

We create the dictionary `ab` using the notation already discussed. We then access key-value pairs by specifying the key using the indexing operator as discussed in the context of lists and tuples. Observe the simple syntax.

We can delete key-value pairs using our old friend - the `del` statement. We simply specify the dictionary and the indexing operator for the key to be removed and pass it to the `del` statement. There is no need to know the value corresponding to the key for this operation.

Next, we access each key-value pair of the dictionary using the `items` method of the dictionary which returns a list of tuples where each tuple contains a pair of items - the key followed by the value. We retrieve this pair and assign it to the variables `name` and `address` correspondingly for each pair using the `for..in` loop and then print these values in the for-block.

We can add new key-value pairs by simply using the indexing operator to access a key and assign that value, as we have done for Guido in the above case.

We can check if a key-value pair exists using the `in` operator.

For the list of methods of the `dict` class, see `help(dict)` .

> **Keyword Arguments and Dictionaries**
>
> If you have used keyword arguments in your functions, you have already used dictionaries! Just think about it - the key-value pair is specified by you in the parameter list of the function definition and when you access variables within your function, it is just a key access of a dictionary (which is called the *symbol table* in compiler design terminology).

Sequence

Lists, tuples and strings are examples of sequences, but what are sequences and what is so special about them?

The major features are *membership tests*, (i.e. the `in` and `not in` expressions) and *indexing operations*, which allow us to fetch a particular item in the sequence directly.

The three types of sequences mentioned above - lists, tuples and strings, also have a *slicing* operation which allows us to retrieve a slice of the sequence i.e. a part of the sequence.

Example (save as `ds_seq.py`):

```
shoplist = ['apple', 'mango', 'carrot', 'banana']
name = 'swaroop'

# Indexing or 'Subscription' operation #
print('Item 0 is', shoplist[0])
print('Item 1 is', shoplist[1])
print('Item 2 is', shoplist[2])
print('Item 3 is', shoplist[3])
print('Item -1 is', shoplist[-1])
print('Item -2 is', shoplist[-2])
print('Character 0 is', name[0])

# Slicing on a list #
print('Item 1 to 3 is', shoplist[1:3])
print('Item 2 to end is', shoplist[2:])
print('Item 1 to -1 is', shoplist[1:-1])
print('Item start to end is', shoplist[:])

# Slicing on a string #
print('characters 1 to 3 is', name[1:3])
print('characters 2 to end is', name[2:])
print('characters 1 to -1 is', name[1:-1])
print('characters start to end is', name[:])
```

Output:

```
$ python ds_seq.py
Item 0 is apple
Item 1 is mango
Item 2 is carrot
Item 3 is banana
Item -1 is banana
Item -2 is carrot
Character 0 is s
Item 1 to 3 is ['mango', 'carrot']
Item 2 to end is ['carrot', 'banana']
Item 1 to -1 is ['mango', 'carrot']
Item start to end is ['apple', 'mango', 'carrot', 'banana']
characters 1 to 3 is wa
characters 2 to end is aroop
characters 1 to -1 is waroo
characters start to end is swaroop
```

How It Works

First, we see how to use indexes to get individual items of a sequence. This is also referred to as the *subscription operation*. Whenever you specify a number to a sequence within square brackets as shown above, Python will fetch you the item corresponding to that

position in the sequence. Remember that Python starts counting numbers from 0. Hence, `shoplist[0]` fetches the first item and `shoplist[3]` fetches the fourth item in the `shoplist` sequence.

The index can also be a negative number, in which case, the position is calculated from the end of the sequence. Therefore, `shoplist[-1]` refers to the last item in the sequence and `shoplist[-2]` fetches the second last item in the sequence.

The slicing operation is used by specifying the name of the sequence followed by an optional pair of numbers separated by a colon within square brackets. Note that this is very similar to the indexing operation you have been using till now. Remember the numbers are optional but the colon isn't.

The first number (before the colon) in the slicing operation refers to the position from where the slice starts and the second number (after the colon) indicates where the slice will stop at. If the first number is not specified, Python will start at the beginning of the sequence. If the second number is left out, Python will stop at the end of the sequence. Note that the slice returned *starts* at the start position and will end just before the *end* position i.e. the start position is included but the end position is excluded from the sequence slice.

Thus, `shoplist[1:3]` returns a slice of the sequence starting at position 1, includes position 2 but stops at position 3 and therefore a *slice* of two items is returned. Similarly, `shoplist[:]` returns a copy of the whole sequence.

You can also do slicing with negative positions. Negative numbers are used for positions from the end of the sequence. For example, `shoplist[:-1]` will return a slice of the sequence which excludes the last item of the sequence but contains everything else.

You can also provide a third argument for the slice, which is the *step* for the slicing (by default, the step size is 1):

```
>>> shoplist = ['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::1]
['apple', 'mango', 'carrot', 'banana']
>>> shoplist[::2]
['apple', 'carrot']
>>> shoplist[::3]
['apple', 'banana']
>>> shoplist[::-1]
['banana', 'carrot', 'mango', 'apple']
```

Notice that when the step is 2, we get the items with position 0, 2,... When the step size is 3, we get the items with position 0, 3, etc.

Try various combinations of such slice specifications using the Python interpreter interactively i.e. the prompt so that you can see the results immediately. The great thing about sequences is that you can access tuples, lists and strings all in the same way!

Set

Sets are *unordered* collections of simple objects. These are used when the existence of an object in a collection is more important than the order or how many times it occurs.

Using sets, you can test for membership, whether it is a subset of another set, find the intersection between two sets, and so on.

```
>>> bri = set(['brazil', 'russia', 'india'])
>>> 'india' in bri
True
>>> 'usa' in bri
False
>>> bric = bri.copy()
>>> bric.add('china')
>>> bric.issuperset(bri)
True
>>> bri.remove('russia')
>>> bri & bric # OR bri.intersection(bric)
{'brazil', 'india'}
```

How It Works

If you remember basic set theory mathematics from school, then this example is fairly self-explanatory. But if not, you can google "set theory" and "Venn diagram" to better understand our use of sets in Python.

References

When you create an object and assign it to a variable, the variable only *refers* to the object and does not represent the object itself! That is, the variable name points to that part of your computer's memory where the object is stored. This is called *binding* the name to the object.

Generally, you don't need to be worried about this, but there is a subtle effect due to references which you need to be aware of:

Example (save as `ds_reference.py`):

```
print('Simple Assignment')
shoplist = ['apple', 'mango', 'carrot', 'banana']
# mylist is just another name pointing to the same object!
mylist = shoplist

# I purchased the first item, so I remove it from the list
del shoplist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that both shoplist and mylist both print
# the same list without the 'apple' confirming that
# they point to the same object

print('Copy by making a full slice')
# Make a copy by doing a full slice
mylist = shoplist[:]
# Remove first item
del mylist[0]

print('shoplist is', shoplist)
print('mylist is', mylist)
# Notice that now the two lists are different
```

Output:

```
$ python ds_reference.py
Simple Assignment
shoplist is ['mango', 'carrot', 'banana']
mylist is ['mango', 'carrot', 'banana']
Copy by making a full slice
shoplist is ['mango', 'carrot', 'banana']
mylist is ['carrot', 'banana']
```

How It Works

Most of the explanation is available in the comments.

Remember that if you want to make a copy of a list or such kinds of sequences or complex objects (not simple *objects* such as integers), then you have to use the slicing operation to make a copy. If you just assign the variable name to another name, both of them will "refer" to the same object and this could be trouble if you are not careful.

> **Note for Perl programmers**
>
> Remember that an assignment statement for lists does **not** create a copy. You have to use slicing operation to make a copy of the sequence.

More About Strings

We have already discussed strings in detail earlier. What more can there be to know? Well, did you know that strings are also objects and have methods which do everything from checking part of a string to stripping spaces? In fact, you've already been using a string method... the `format` method!

The strings that you use in programs are all objects of the class `str`. Some useful methods of this class are demonstrated in the next example. For a complete list of such methods, see `help(str)`.

Example (save as `ds_str_methods.py`):

```
# This is a string object
name = 'Swaroop'

if name.startswith('Swa'):
    print('Yes, the string starts with "Swa"')

if 'a' in name:
    print('Yes, it contains the string "a"')

if name.find('war') != -1:
    print('Yes, it contains the string "war"')

delimiter = '_*_'
mylist = ['Brazil', 'Russia', 'India', 'China']
print(delimiter.join(mylist))
```

Output:

```
$ python ds_str_methods.py
Yes, the string starts with "Swa"
Yes, it contains the string "a"
Yes, it contains the string "war"
Brazil_*_Russia_*_India_*_China
```

How It Works

Here, we see a lot of the string methods in action. The `startswith` method is used to find out whether the string starts with the given string. The `in` operator is used to check if a given string is a part of the string.

The `find` method is used to locate the position of the given substring within the string; `find` returns -1 if it is unsuccessful in finding the substring. The `str` class also has a neat method to `join` the items of a sequence with the string acting as a delimiter between each

item of the sequence and returns a bigger string generated from this.

Summary

We have explored the various built-in data structures of Python in detail. These data structures will be essential for writing programs of reasonable size.

Now that we have a lot of the basics of Python in place, we will next see how to design and write a real-world Python program.

Problem Solving

We have explored various parts of the Python language and now we will take a look at how all these parts fit together, by designing and writing a program which *does* something useful. The idea is to learn how to write a Python script on your own.

The Problem

The problem we want to solve is:

> I want a program which creates a backup of all my important files.

Although, this is a simple problem, there is not enough information for us to get started with the solution. A little more *analysis* is required. For example, how do we specify *which* files are to be backed up? *How* are they stored? *Where* are they stored?

After analyzing the problem properly, we *design* our program. We make a list of things about how our program should work. In this case, I have created the following list on how *I* want it to work. If you do the design, you may not come up with the same kind of analysis since every person has their own way of doing things, so that is perfectly okay.

- The files and directories to be backed up are specified in a list.
- The backup must be stored in a main backup directory.
- The files are backed up into a zip file.
- The name of the zip archive is the current date and time.
- We use the standard `zip` command available by default in any standard GNU/Linux or Unix distribution. Note that you can use any archiving command you want as long as it has a command line interface.

> **For Windows users**
>
> Windows users can install the `zip` command from the GnuWin32 project page and add `C:\Program Files\GnuWin32\bin` to your system `PATH` environment variable, similar to what we did for recognizing the python command itself.

The Solution

As the design of our program is now reasonably stable, we can write the code which is an *implementation* of our solution.

Save as `backup_ver1.py` :

```
import os
import time

# 1. The files and directories to be backed up are
# specified in a list.
# Example on Windows:
# source = ['"C:\\My Documents"']
# Example on Mac OS X and Linux:
source = ['/Users/swa/notes']
# Notice we have to use double quotes inside a string
# for names with spaces in it.  We could have also used
# a raw string by writing [r'C:\My Documents'].

# 2. The backup must be stored in a
# main backup directory
# Example on Windows:
# target_dir = 'E:\\Backup'
# Example on Mac OS X and Linux:
target_dir = '/Users/swa/backup'
# Remember to change this to which folder you will be using

# 3. The files are backed up into a zip file.
# 4. The name of the zip archive is the current date and time
target = target_dir + os.sep + \
         time.strftime('%Y%m%d%H%M%S') + '.zip'

# Create target directory if it is not present
if not os.path.exists(target_dir):
    os.mkdir(target_dir)  # make directory

# 5. We use the zip command to put the files in a zip archive
zip_command = 'zip -r {0} {1}'.format(target,
                                      ' '.join(source))

# Run the backup
print('Zip command is:')
print(zip_command)
print('Running:')
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

Output:

```
$ python backup_ver1.py
Zip command is:
zip -r /Users/swa/backup/20140328084844.zip /Users/swa/notes
Running:
  adding: Users/swa/notes/ (stored 0%)
  adding: Users/swa/notes/blah1.txt (stored 0%)
  adding: Users/swa/notes/blah2.txt (stored 0%)
  adding: Users/swa/notes/blah3.txt (stored 0%)
Successful backup to /Users/swa/backup/20140328084844.zip
```

Now, we are in the *testing* phase where we test that our program works properly. If it doesn't behave as expected, then we have to *debug* our program i.e. remove the *bugs* (errors) from the program.

If the above program does not work for you, copy the line printed after the `Zip command is` line in the output, paste it in the shell (on GNU/Linux and Mac OS X) / `cmd` (on Windows), see what the error is and try to fix it. Also check the zip command manual on what could be wrong. If this command succeeds, then the problem might be in the Python program itself, so check if it exactly matches the program written above.

How It Works

You will notice how we have converted our *design* into *code* in a step-by-step manner.

We make use of the `os` and `time` modules by first importing them. Then, we specify the files and directories to be backed up in the `source` list. The target directory is where we store all the backup files and this is specified in the `target_dir` variable. The name of the zip archive that we are going to create is the current date and time which we generate using the `time.strftime()` function. It will also have the `.zip` extension and will be stored in the `target_dir` directory.

Notice the use of the `os.sep` variable - this gives the directory separator according to your operating system, i.e. it will be `'/'` in GNU/Linux, Unix, macOS, and will be `'\\'` in Windows. Using `os.sep` instead of these characters directly will make our program portable and work across all of these systems.

The `time.strftime()` function takes a specification such as the one we have used in the above program. The `%Y` specification will be replaced by the year with the century. The `%m` specification will be replaced by the month as a decimal number between `01` and `12` and so on. The complete list of such specifications can be found in the Python Reference Manual.

We create the name of the target zip file using the addition operator which *concatenates* the strings i.e. it joins the two strings together and returns a new one. Then, we create a string `zip_command` which contains the command that we are going to execute. You can check if

this command works by running it in the shell (GNU/Linux terminal or DOS prompt).

The `zip` command that we are using has some options available, and one of these options is `-r`. The `-r` option specifies that the zip command should work **r**ecursively for directories, i.e. it should include all the subdirectories and files. Options are followed by the name of the zip archive to create, followed by the list of files and directories to backup. We convert the `source` list into a string using the `join` method of strings which we have already seen how to use.

Then, we finally *run* the command using the `os.system` function which runs the command as if it was run from the *system* i.e. in the shell - it returns `0` if the command was successfully, else it returns an error number.

Depending on the outcome of the command, we print the appropriate message that the backup has failed or succeeded.

That's it, we have created a script to take a backup of our important files!

> **Note to Windows Users**
>
> Instead of double backslash escape sequences, you can also use raw strings. For example, use `'C:\\Documents'` or `r'C:\Documents'`. However, do *not* use `'C:\Documents'` since you end up using an unknown escape sequence `\D`.

Now that we have a working backup script, we can use it whenever we want to take a backup of the files. This is called the *operation* phase or the *deployment* phase of the software.

The above program works properly, but (usually) first programs do not work exactly as you expect. For example, there might be problems if you have not designed the program properly or if you have made a mistake when typing the code, etc. Appropriately, you will have to go back to the design phase or you will have to debug your program.

Second Version

The first version of our script works. However, we can make some refinements to it so that it can work better on a daily basis. This is called the *maintenance* phase of the software.

One of the refinements I felt was useful is a better file-naming mechanism - using the *time* as the name of the file within a directory with the current *date* as a directory within the main backup directory. The first advantage is that your backups are stored in a hierarchical manner and therefore it is much easier to manage. The second advantage is that the

filenames are much shorter. The third advantage is that separate directories will help you check if you have made a backup for each day since the directory would be created only if you have made a backup for that day.

Save as `backup_ver2.py` :

```
import os
import time

# 1. The files and directories to be backed up are
# specified in a list.
# Example on Windows:
# source = ['"C:\\My Documents"', 'C:\\Code']
# Example on Mac OS X and Linux:
source = ['/Users/swa/notes']
# Notice we had to use double quotes inside the string
# for names with spaces in it.

# 2. The backup must be stored in a
# main backup directory
# Example on Windows:
# target_dir = 'E:\\Backup'
# Example on Mac OS X and Linux:
target_dir = '/Users/swa/backup'
# Remember to change this to which folder you will be using

# Create target directory if it is not present
if not os.path.exists(target_dir):
    os.mkdir(target_dir)  # make directory

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory
# in the main directory.
today = target_dir + os.sep + time.strftime('%Y%m%d')
# The current time is the name of the zip archive.
now = time.strftime('%H%M%S')

# The name of the zip file
target = today + os.sep + now + '.zip'

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today)
    print('Successfully created directory', today)

# 5. We use the zip command to put the files in a zip archive
zip_command = 'zip -r {0} {1}'.format(target,
                                      ' '.join(source))

# Run the backup
print('Zip command is:')
print(zip_command)
print('Running:')
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

Output:

```
$ python backup_ver2.py
Successfully created directory /Users/swa/backup/20140329
Zip command is:
zip -r /Users/swa/backup/20140329/073201.zip /Users/swa/notes
Running:
  adding: Users/swa/notes/ (stored 0%)
  adding: Users/swa/notes/blah1.txt (stored 0%)
  adding: Users/swa/notes/blah2.txt (stored 0%)
  adding: Users/swa/notes/blah3.txt (stored 0%)
Successful backup to /Users/swa/backup/20140329/073201.zip
```

How It Works

Most of the program remains the same. The changes are that we check if there is a directory with the current day as its name inside the main backup directory using the `os.path.exists` function. If it doesn't exist, we create it using the `os.mkdir` function.

Third Version

The second version works fine when I do many backups, but when there are lots of backups, I am finding it hard to differentiate what the backups were for! For example, I might have made some major changes to a program or presentation, then I want to associate what those changes are with the name of the zip archive. This can be easily achieved by attaching a user-supplied comment to the name of the zip archive.

WARNING: The following program does not work, so do not be alarmed, please follow along because there's a lesson in here.

Save as `backup_ver3.py` :

```
import os
import time

# 1. The files and directories to be backed up are
# specified in a list.
# Example on Windows:
# source = ['"C:\\My Documents"', 'C:\\Code']
# Example on Mac OS X and Linux:
source = ['/Users/swa/notes']
# Notice we had to use double quotes inside the string
# for names with spaces in it.

# 2. The backup must be stored in a
# main backup directory
```

```
# Example on Windows:
# target_dir = 'E:\\Backup'
# Example on Mac OS X and Linux:
target_dir = '/Users/swa/backup'
# Remember to change this to which folder you will be using

# Create target directory if it is not present
if not os.path.exists(target_dir):
    os.mkdir(target_dir)  # make directory

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory
# in the main directory.
today = target_dir + os.sep + time.strftime('%Y%m%d')
# The current time is the name of the zip archive.
now = time.strftime('%H%M%S')

# Take a comment from the user to
# create the name of the zip file
comment = input('Enter a comment --> ')
# Check if a comment was entered
if len(comment) == 0:
    target = today + os.sep + now + '.zip'
else:
    target = today + os.sep + now + '_' +
        comment.replace(' ', '_') + '.zip'

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today)
    print('Successfully created directory', today)

# 5. We use the zip command to put the files in a zip archive
zip_command = "zip -r {0} {1}".format(target,
                                      ' '.join(source))

# Run the backup
print('Zip command is:')
print(zip_command)
print('Running:')
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

Output:

```
$ python backup_ver3.py
  File "backup_ver3.py", line 39
    target = today + os.sep + now + '_' +
                                        ^
SyntaxError: invalid syntax
```

How This (does not) Work

This program does not work! Python says there is a syntax error which means that the script does not satisfy the structure that Python expects to see. When we observe the error given by Python, it also tells us the place where it detected the error as well. So we start *debugging* our program from that line.

On careful observation, we see that the single logical line has been split into two physical lines but we have not specified that these two physical lines belong together. Basically, Python has found the addition operator (`+`) without any operand in that logical line and hence it doesn't know how to continue. Remember that we can specify that the logical line continues in the next physical line by the use of a backslash at the end of the physical line. So, we make this correction to our program. This correction of the program when we find errors is called *bug fixing*.

Fourth Version

Save as `backup_ver4.py` :

```
import os
import time

# 1. The files and directories to be backed up are
# specified in a list.
# Example on Windows:
# source = ['"C:\\My Documents"', 'C:\\Code']
# Example on Mac OS X and Linux:
source = ['/Users/swa/notes']
# Notice we had to use double quotes inside the string
# for names with spaces in it.

# 2. The backup must be stored in a
# main backup directory
# Example on Windows:
# target_dir = 'E:\\Backup'
# Example on Mac OS X and Linux:
target_dir = '/Users/swa/backup'
# Remember to change this to which folder you will be using

# Create target directory if it is not present
```

```
if not os.path.exists(target_dir):
    os.mkdir(target_dir)  # make directory

# 3. The files are backed up into a zip file.
# 4. The current day is the name of the subdirectory
# in the main directory.
today = target_dir + os.sep + time.strftime('%Y%m%d')
# The current time is the name of the zip archive.
now = time.strftime('%H%M%S')

# Take a comment from the user to
# create the name of the zip file
comment = input('Enter a comment --> ')
# Check if a comment was entered
if len(comment) == 0:
    target = today + os.sep + now + '.zip'
else:
    target = today + os.sep + now + '_' + \
        comment.replace(' ', '_') + '.zip'

# Create the subdirectory if it isn't already there
if not os.path.exists(today):
    os.mkdir(today)
    print('Successfully created directory', today)

# 5. We use the zip command to put the files in a zip archive
zip_command = 'zip -r {0} {1}'.format(target,
                                      ' '.join(source))

# Run the backup
print('Zip command is:')
print(zip_command)
print('Running:')
if os.system(zip_command) == 0:
    print('Successful backup to', target)
else:
    print('Backup FAILED')
```

Output:

```
$ python backup_ver4.py
Enter a comment --> added new examples
Zip command is:
zip -r /Users/swa/backup/20140329/074122_added_new_examples.zip /Users/swa/notes
Running:
  adding: Users/swa/notes/ (stored 0%)
  adding: Users/swa/notes/blah1.txt (stored 0%)
  adding: Users/swa/notes/blah2.txt (stored 0%)
  adding: Users/swa/notes/blah3.txt (stored 0%)
Successful backup to /Users/swa/backup/20140329/074122_added_new_examples.zip
```

How It Works

This program now works! Let us go through the actual enhancements that we had made in version 3. We take in the user's comments using the `input` function and then check if the user actually entered something by finding out the length of the input using the `len` function. If the user has just pressed `enter` without entering anything (maybe it was just a routine backup or no special changes were made), then we proceed as we have done before.

However, if a comment was supplied, then this is attached to the name of the zip archive just before the `.zip` extension. Notice that we are replacing spaces in the comment with underscores - this is because managing filenames without spaces is much easier.

More Refinements

The fourth version is a satisfactorily working script for most users, but there is always room for improvement. For example, you can include a *verbosity* level for the zip command by specifying a `-v` option to make your program become more talkative or a `-q` option to make it *quiet*.

Another possible enhancement would be to allow extra files and directories to be passed to the script at the command line. We can get these names from the `sys.argv` list and we can add them to our `source` list using the `extend` method provided by the `list` class.

The most important refinement would be to not use the `os.system` way of creating archives and instead using the zipfile or tarfile built-in modules to create these archives. They are part of the standard library and available already for you to use without external dependencies on the zip program to be available on your computer.

However, I have been using the `os.system` way of creating a backup in the above examples purely for pedagogical purposes, so that the example is simple enough to be understood by everybody but real enough to be useful.

Can you try writing the fifth version that uses the zipfile module instead of the `os.system` call?

The Software Development Process

We have now gone through the various *phases* in the process of writing a software. These phases can be summarised as follows:

1. What (Analysis)

2. How (Design)
3. Do It (Implementation)
4. Test (Testing and Debugging)
5. Use (Operation or Deployment)
6. Maintain (Refinement)

A recommended way of writing programs is the procedure we have followed in creating the backup script: Do the analysis and design. Start implementing with a simple version. Test and debug it. Use it to ensure that it works as expected. Now, add any features that you want and continue to repeat the Do It-Test-Use cycle as many times as required.

Remember:

> Software is grown, not built. -- Bill de hÓra

Summary

We have seen how to create our own Python programs/scripts and the various stages involved in writing such programs. You may find it useful to create your own program just like we did in this chapter so that you become comfortable with Python as well as problem-solving.

Next, we will discuss object-oriented programming.

Object Oriented Programming

In all the programs we wrote till now, we have designed our program around functions i.e. blocks of statements which manipulate data. This is called the *procedure-oriented* way of programming. There is another way of organizing your program which is to combine data and functionality and wrap it inside something called an object. This is called the *object oriented* programming paradigm. Most of the time you can use procedural programming, but when writing large programs or have a problem that is better suited to this method, you can use object oriented programming techniques.

Classes and objects are the two main aspects of object oriented programming. A **class** creates a new *type* where **objects** are **instances** of the class. An analogy is that you can have variables of type `int` which translates to saying that variables that store integers are variables which are instances (objects) of the `int` class.

> **Note for Static Language Programmers**
>
> Note that even integers are treated as objects (of the `int` class). This is unlike C++ and Java (before version 1.5) where integers are primitive native types.
>
> See `help(int)` for more details on the class.
>
> C# and Java 1.5 programmers will find this similar to the *boxing and unboxing* concept.

Objects can store data using ordinary variables that *belong* to the object. Variables that belong to an object or class are referred to as **fields**. Objects can also have functionality by using functions that *belong* to a class. Such functions are called **methods** of the class. This terminology is important because it helps us to differentiate between functions and variables which are independent and those which belong to a class or object. Collectively, the fields and methods can be referred to as the **attributes** of that class.

Fields are of two types - they can belong to each instance/object of the class or they can belong to the class itself. They are called **instance variables** and **class variables** respectively.

A class is created using the `class` keyword. The fields and methods of the class are listed in an indented block.

The `self`

Class methods have only one specific difference from ordinary functions - they must have an extra first name that has to be added to the beginning of the parameter list, but you **do not** give a value for this parameter when you call the method, Python will provide it. This particular variable refers to the object *itself*, and by convention, it is given the name `self`.

Although, you can give any name for this parameter, it is *strongly recommended* that you use the name `self` - any other name is definitely frowned upon. There are many advantages to using a standard name - any reader of your program will immediately recognize it and even specialized IDEs (Integrated Development Environments) can help you if you use `self`.

> **Note for C++/Java/C# Programmers**
>
> The `self` in Python is equivalent to the `this` pointer in C++ and the `this` reference in Java and C#.

You must be wondering how Python gives the value for `self` and why you don't need to give a value for it. An example will make this clear. Say you have a class called `MyClass` and an instance of this class called `myobject`. When you call a method of this object as `myobject.method(arg1, arg2)`, this is automatically converted by Python into `MyClass.method(myobject, arg1, arg2)` - this is all the special `self` is about.

This also means that if you have a method which takes no arguments, then you still have to have one argument - the `self`.

Classes

The simplest class possible is shown in the following example (save as `oop_simplestclass.py`).

```
class Person:
    pass  # An empty block

p = Person()
print(p)
```

Output:

```
$ python oop_simplestclass.py
<__main__.Person instance at 0x10171f518>
```

How It Works

We create a new class using the `class` statement and the name of the class. This is followed by an indented block of statements which form the body of the class. In this case, we have an empty block which is indicated using the `pass` statement.

Next, we create an object/instance of this class using the name of the class followed by a pair of parentheses. (We will learn more about instantiation in the next section). For our verification, we confirm the type of the variable by simply printing it. It tells us that we have an instance of the `Person` class in the `__main__` module.

Notice that the address of the computer memory where your object is stored is also printed. The address will have a different value on your computer since Python can store the object wherever it finds space.

Methods

We have already discussed that classes/objects can have methods just like functions except that we have an extra `self` variable. We will now see an example (save as `oop_method.py`).

```
class Person:
    def say_hi(self):
        print('Hello, how are you?')

p = Person()
p.say_hi()
# The previous 2 lines can also be written as
# Person().say_hi()
```

Output:

```
$ python oop_method.py
Hello, how are you?
```

How It Works

Here we see the `self` in action. Notice that the `say_hi` method takes no parameters but still has the `self` in the function definition.

The `__init__` method

There are many method names which have special significance in Python classes. We will see the significance of the `__init__` method now.

The `__init__` method is run as soon as an object of a class is instantiated (i.e. created). The method is useful to do any *initialization* (i.e. passing initial values to your object) you want to do with your object. Notice the double underscores both at the beginning and at the end of the name.

Example (save as `oop_init.py`):

```
class Person:
    def __init__(self, name):
        self.name = name

    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Swaroop')
p.say_hi()
# The previous 2 lines can also be written as
# Person('Swaroop').say_hi()
```

Output:

```
$ python oop_init.py
Hello, my name is Swaroop
```

How It Works

Here, we define the `__init__` method as taking a parameter `name` (along with the usual `self`). Here, we just create a new field also called `name`. Notice these are two different variables even though they are both called 'name'. There is no problem because the dotted notation `self.name` means that there is something called "name" that is part of the object called "self" and the other `name` is a local variable. Since we explicitly indicate which name we are referring to, there is no confusion.

When creating new instance `p`, of the class `Person`, we do so by using the class name, followed by the arguments in the parentheses: p = Person('Swaroop').

We do not explicitly call the `__init__` method. This is the special significance of this method.

Now, we are able to use the `self.name` field in our methods which is demonstrated in the `say_hi` method.

Class And Object Variables

We have already discussed the functionality part of classes and objects (i.e. methods), now let us learn about the data part. The data part, i.e. fields, are nothing but ordinary variables that are *bound* to the **namespaces** of the classes and objects. This means that these names are valid within the context of these classes and objects only. That's why they are called *name spaces*.

There are two types of *fields* - class variables and object variables which are classified depending on whether the class or the object *owns* the variables respectively.

Class variables are shared - they can be accessed by all instances of that class. There is only one copy of the class variable and when any one object makes a change to a class variable, that change will be seen by all the other instances.

Object variables are owned by each individual object/instance of the class. In this case, each object has its own copy of the field i.e. they are not shared and are not related in any way to the field by the same name in a different instance. An example will make this easy to understand (save as `oop_objvar.py`):

```
class Robot:
    """Represents a robot, with a name."""

    # A class variable, counting the number of robots
    population = 0

    def __init__(self, name):
        """Initializes the data."""
        self.name = name
        print("(Initializing {})".format(self.name))

        # When this person is created, the robot
        # adds to the population
        Robot.population += 1

    def die(self):
        """I am dying."""
        print("{} is being destroyed!".format(self.name))

        Robot.population -= 1

        if Robot.population == 0:
            print("{} was the last one.".format(self.name))
        else:
            print("There are still {:d} robots working.".format(
                Robot.population))

    def say_hi(self):
        """Greeting by the robot.

        Yeah, they can do that."""
```

```
        print("Greetings, my masters call me {}.".format(self.name))

    @classmethod
    def how_many(cls):
        """Prints the current population."""
        print("We have {:d} robots.".format(cls.population))


droid1 = Robot("R2-D2")
droid1.say_hi()
Robot.how_many()

droid2 = Robot("C-3PO")
droid2.say_hi()
Robot.how_many()

print("\nRobots can do some work here.\n")

print("Robots have finished their work. So let's destroy them.")
droid1.die()
droid2.die()

Robot.how_many()
```

Output:

```
$ python oop_objvar.py
(Initializing R2-D2)
Greetings, my masters call me R2-D2.
We have 1 robots.
(Initializing C-3PO)
Greetings, my masters call me C-3PO.
We have 2 robots.

Robots can do some work here.

Robots have finished their work. So let's destroy them.
R2-D2 is being destroyed!
There are still 1 robots working.
C-3PO is being destroyed!
C-3PO was the last one.
We have 0 robots.
```

How It Works

This is a long example but helps demonstrate the nature of class and object variables. Here, `population` belongs to the `Robot` class and hence is a class variable. The `name` variable belongs to the object (it is assigned using `self`) and hence is an object variable.

Thus, we refer to the `population` class variable as `Robot.population` and not as `self.population`. We refer to the object variable `name` using `self.name` notation in the methods of that object. Remember this simple difference between class and object variables. Also note that an object variable with the same name as a class variable will hide the class variable!

Instead of `Robot.population`, we could have also used `self.__class__.population` because every object refers to its class via the `self.__class__` attribute.

The `how_many` is actually a method that belongs to the class and not to the object. This means we can define it as either a `classmethod` or a `staticmethod` depending on whether we need to know which class we are part of. Since we refer to a class variable, let's use `classmethod`.

We have marked the `how_many` method as a class method using a decorator.

Decorators can be imagined to be a shortcut to calling a wrapper function (i.e. a function that "wraps" around another function so that it can do something before or after the inner function), so applying the `@classmethod` decorator is the same as calling:

```
how_many = classmethod(how_many)
```

Observe that the `__init__` method is used to initialize the `Robot` instance with a name. In this method, we increase the `population` count by 1 since we have one more robot being added. Also observe that the values of `self.name` is specific to each object which indicates the nature of object variables.

Remember, that you must refer to the variables and methods of the same object using the `self` *only*. This is called an *attribute reference*.

In this program, we also see the use of *docstrings* for classes as well as methods. We can access the class docstring at runtime using `Robot.__doc__` and the method docstring as `Robot.say_hi.__doc__`

In the `die` method, we simply decrease the `Robot.population` count by 1.

All class members are public. One exception: If you use data members with names using the *double underscore prefix* such as `__privatevar`, Python uses name-mangling to effectively make it a private variable.

Thus, the convention followed is that any variable that is to be used only within the class or object should begin with an underscore and all other names are public and can be used by other classes/objects. Remember that this is only a convention and is not enforced by Python (except for the double underscore prefix).

> **Note for C++/Java/C# Programmers**
>
> All class members (including the data members) are *public* and all the methods are *virtual* in Python.

Inheritance

One of the major benefits of object oriented programming is **reuse** of code and one of the ways this is achieved is through the **inheritance** mechanism. Inheritance can be best imagined as implementing a **type and subtype** relationship between classes.

Suppose you want to write a program which has to keep track of the teachers and students in a college. They have some common characteristics such as name, age and address. They also have specific characteristics such as salary, courses and leaves for teachers and, marks and fees for students.

You can create two independent classes for each type and process them but adding a new common characteristic would mean adding to both of these independent classes. This quickly becomes unwieldy.

A better way would be to create a common class called `SchoolMember` and then have the teacher and student classes *inherit* from this class, i.e. they will become sub-types of this type (class) and then we can add specific characteristics to these sub-types.

There are many advantages to this approach. If we add/change any functionality in `SchoolMember`, this is automatically reflected in the subtypes as well. For example, you can add a new ID card field for both teachers and students by simply adding it to the SchoolMember class. However, changes in the subtypes do not affect other subtypes. Another advantage is that you can refer to a teacher or student object as a `SchoolMember` object which could be useful in some situations such as counting of the number of school members. This is called **polymorphism** where a sub-type can be substituted in any situation where a parent type is expected, i.e. the object can be treated as an instance of the parent class.

Also observe that we reuse the code of the parent class and we do not need to repeat it in the different classes as we would have had to in case we had used independent classes.

The `SchoolMember` class in this situation is known as the **base class** or the **superclass**. The `Teacher` and `Student` classes are called the **derived classes** or **subclasses**.

We will now see this example as a program (save as `oop_subclass.py`):

```
class SchoolMember:
    '''Represents any school member.'''
    def __init__(self, name, age):
        self.name = name
        self.age = age
        print('(Initialized SchoolMember: {})'.format(self.name))

    def tell(self):
        '''Tell my details.'''
        print('Name:"{}" Age:"{}"'.format(self.name, self.age), end=" ")


class Teacher(SchoolMember):
    '''Represents a teacher.'''
    def __init__(self, name, age, salary):
        SchoolMember.__init__(self, name, age)
        self.salary = salary
        print('(Initialized Teacher: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Salary: "{:d}"'.format(self.salary))


class Student(SchoolMember):
    '''Represents a student.'''
    def __init__(self, name, age, marks):
        SchoolMember.__init__(self, name, age)
        self.marks = marks
        print('(Initialized Student: {})'.format(self.name))

    def tell(self):
        SchoolMember.tell(self)
        print('Marks: "{:d}"'.format(self.marks))

t = Teacher('Mrs. Shrividya', 40, 30000)
s = Student('Swaroop', 25, 75)

# prints a blank line
print()

members = [t, s]
for member in members:
    # Works for both Teachers and Students
    member.tell()
```

Output:

```
$ python oop_subclass.py
(Initialized SchoolMember: Mrs. Shrividya)
(Initialized Teacher: Mrs. Shrividya)
(Initialized SchoolMember: Swaroop)
(Initialized Student: Swaroop)

Name:"Mrs. Shrividya" Age:"40" Salary: "30000"
Name:"Swaroop" Age:"25" Marks: "75"
```

How It Works

To use inheritance, we specify the base class names in a tuple following the class name in the class definition (for example, `class Teacher(SchoolMember)`). Next, we observe that the `__init__` method of the base class is explicitly called using the `self` variable so that we can initialize the base class part of an instance in the subclass. This is very important to remember- Since we are defining a `__init__` method in `Teacher` and `Student` subclasses, Python does not automatically call the constructor of the base class `SchoolMember` , you have to explicitly call it yourself.

In contrast, if we have not defined an `__init__` method in a subclass, Python will call the constructor of the base class automatically.

While we could treat instances of `Teacher` or `Student` as we would an instance of `SchoolMember` and access the `tell` method of `SchoolMember` by simply typing `Teacher.tell` or `Student.tell` , we instead define another `tell` method in each subclass (using the `tell` method of `SchoolMember` for part of it) to tailor it for that subclass. Because we have done this, when we write `Teacher.tell` Python uses the `tell` method for that subclass vs the superclass. However, if we did not have a `tell` method in the subclass, Python would use the `tell` method in the superclass. Python always starts looking for methods in the actual subclass type first, and if it doesn't find anything, it starts looking at the methods in the subclass's base classes, one by one in the order they are specified in the tuple (here we only have 1 base class, but you can have multiple base classes) in the class definition.

A note on terminology - if more than one class is listed in the inheritance tuple, then it is called **multiple inheritance**.

The `end` parameter is used in the `print` function in the superclass's `tell()` method to print a line and allow the next print to continue on the same line. This is a trick to make `print` not print a `\n` (newline) symbol at the end of the printing.

Summary

We have now explored the various aspects of classes and objects as well as the various terminologies associated with it. We have also seen the benefits and pitfalls of object-oriented programming. Python is highly object-oriented and understanding these concepts carefully will help you a lot in the long run.

Next, we will learn how to deal with input/output and how to access files in Python.

Input and Output

There will be situations where your program has to interact with the user. For example, you would want to take input from the user and then print some results back. We can achieve this using the `input()` function and `print` function respectively.

For output, we can also use the various methods of the `str` (string) class. For example, you can use the `rjust` method to get a string which is right justified to a specified width. See `help(str)` for more details.

Another common type of input/output is dealing with files. The ability to create, read and write files is essential to many programs and we will explore this aspect in this chapter.

Input from user

Save this program as `io_input.py` :

```
def reverse(text):
    return text[::-1]


def is_palindrome(text):
    return text == reverse(text)


something = input("Enter text: ")
if is_palindrome(something):
    print("Yes, it is a palindrome")
else:
    print("No, it is not a palindrome")
```

Output:

```
$ python3 io_input.py
Enter text: sir
No, it is not a palindrome

$ python3 io_input.py
Enter text: madam
Yes, it is a palindrome

$ python3 io_input.py
Enter text: racecar
Yes, it is a palindrome
```

How It Works

We use the slicing feature to reverse the text. We've already seen how we can make slices from sequences using the `seq[a:b]` code starting from position `a` to position `b`. We can also provide a third argument that determines the *step* by which the slicing is done. The default step is `1` because of which it returns a continuous part of the text. Giving a negative step, i.e., `-1` will return the text in reverse.

The `input()` function takes a string as argument and displays it to the user. Then it waits for the user to type something and press the return key. Once the user has entered and pressed the return key, the `input()` function will then return that text the user has entered.

We take that text and reverse it. If the original text and reversed text are equal, then the text is a palindrome.

Homework exercise

Checking whether a text is a palindrome should also ignore punctuation, spaces and case. For example, "Rise to vote, sir." is also a palindrome but our current program doesn't say it is. Can you improve the above program to recognize this palindrome?

If you need a hint, the idea is that...[1]

Files

You can open and use files for reading or writing by creating an object of the `file` class and using its `read`, `readline` or `write` methods appropriately to read from or write to the file. The ability to read or write to the file depends on the mode you have specified for the file opening. Then finally, when you are finished with the file, you call the `close` method to tell Python that we are done using the file.

Example (save as `io_using_file.py`):

```
poem = '''\
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
'''

# Open for 'w'riting
f = open('poem.txt', 'w')
# Write text to file
f.write(poem)
# Close the file
f.close()

# If no mode is specified,
# 'r'ead mode is assumed by default
f = open('poem.txt')
while True:
    line = f.readline()
    # Zero length indicates EOF
    if len(line) == 0:
        break
    # The `line` already has a newline
    # at the end of each line
    # since it is reading from a file.
    print(line, end='')
# close the file
f.close()
```

Output:

```
$ python3 io_using_file.py
Programming is fun
When the work is done
if you wanna make your work also fun:
    use Python!
```

How It Works

Note that we can create a new file object simply by using the `open` method. We open (or create it if it doesn't already exist) this file by using the built-in `open` function and specifying the name of the file and the mode in which we want to open the file. The mode can be a read mode (`'r'`), write mode (`'w'`) or append mode (`'a'`). We can also specify whether we are reading, writing, or appending in text mode (`'t'`) or binary mode (`'b'`). There are actually many more modes available and `help(open)` will give you more details about them. By default, `open()` considers the file to be a 't'ext file and opens it in 'r'ead mode.

In our example, we first open/create the file in write text mode and use the `write` method of the file object to write our string variable `poem` to the file and then we finally `close` the file.

Next, we open the same file again for reading. We don't need to specify a mode because 'read text file' is the default mode. We read in each line of the file using the `readline` method in a loop. This method returns a complete line including the newline character at the end of the line. When an *empty* string is returned, it means that we have reached the end of the file and we 'break' out of the loop.

In the end, we finally `close` the file.

We can see from our `readline` output that this program has indeed written to and read from our new `poem.txt` file.

Pickle

Python provides a standard module called `pickle` which you can use to store *any* plain Python object in a file and then get it back later. This is called storing the object *persistently*.

Example (save as `io_pickle.py`):

```
import pickle

# The name of the file where we will store the object
shoplistfile = 'shoplist.data'
# The list of things to buy
shoplist = ['apple', 'mango', 'carrot']

# Write to the file
f = open(shoplistfile, 'wb')
# Dump the object to a file
pickle.dump(shoplist, f)
f.close()

# Destroy the shoplist variable
del shoplist

# Read back from the storage
f = open(shoplistfile, 'rb')
# Load the object from the file
storedlist = pickle.load(f)
print(storedlist)
```

Output:

```
$ python io_pickle.py
['apple', 'mango', 'carrot']
```

How It Works

To store an object in a file, we have to first `open` the file in **w**rite **b**inary mode and then call the `dump` function of the `pickle` module. This process is called *pickling*.

Next, we retrieve the object using the `load` function of the `pickle` module which returns the object. This process is called *unpickling*.

Unicode

So far, when we have been writing and using strings, or reading and writing to a file, we have used simple English characters only. Both English and non-English characters can be represented in Unicode (please see the articles at the end of this section for more info), and Python 3 by default stores string variables (think of all that text we wrote using single or double or triple quotes) in Unicode.

> NOTE: If you are using Python 2, and we want to be able to read and write other non-English languages, we need to use the `unicode` type, and it all starts with the character `u`, e.g. `u"hello world"`

```
>>> "hello world"
'hello world'
>>> type("hello world")
<class 'str'>
>>> u"hello world"
'hello world'
>>> type(u"hello world")
<class 'str'>
```

When data is sent over the Internet, we need to send it in bytes... something your computer easily understands. The rules for translating Unicode (which is what Python uses when it stores a string) to bytes is called encoding. A popular encoding to use is UTF-8. We can read and write in UTF-8 by using a simple keyword argument in our `open` function.

```
# encoding=utf-8
import io

f = io.open("abc.txt", "wt", encoding="utf-8")
f.write(u"Imagine non-English language here")
f.close()

text = io.open("abc.txt", encoding="utf-8").read()
print(text)
```

How It Works

We use `io.open` and then use the `encoding` argument in the first open statement to encode the message, and then again in the second open statement when decoding the message. Note that we should only use encoding in the open statement when in text mode.

Whenever we write a program that uses Unicode literals (by putting a `u` before the string) like we have used above, we have to make sure that Python itself is told that our program uses UTF-8, and we have to put `# encoding=utf-8` comment at the top of our program.

You should learn more about this topic by reading:

- "The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets"
- Python Unicode Howto
- Pragmatic Unicode talk by Nat Batchelder

Summary

We have discussed various types of input/output, about file handling, about the pickle module and about Unicode.

Next, we will explore the concept of exceptions.

> 1. Use a tuple (you can find a list of *all* punctuation marks here) to hold all the forbidden characters, then use the membership test to determine whether a character should be removed or not, i.e. forbidden = (`!` , `?` , `.` , ...). ↩

Exceptions

Exceptions occur when *exceptional* situations occur in your program. For example, what if you are going to read a file and the file does not exist? Or what if you accidentally deleted it when the program was running? Such situations are handled using **exceptions**.

Similarly, what if your program had some invalid statements? This is handled by Python which **raises** its hands and tells you there is an **error**.

Errors

Consider a simple `print` function call. What if we misspelt `print` as `Print`? Note the capitalization. In this case, Python *raises* a syntax error.

```
>>> Print("Hello World")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Print' is not defined
>>> print("Hello World")
Hello World
```

Observe that a `NameError` is raised and also the location where the error was detected is printed. This is what an **error handler** for this error does.

Exceptions

We will **try** to read input from the user. Enter the first line below and hit the `Enter` key. When your computer prompts you for input, instead press `[ctrl-d]` on a Mac or `[ctrl-z]` with Windows and see what happens. (If you're using Windows and neither option works, you can try `[ctrl-c]` in the Command Prompt to generate a KeyboardInterrupt error instead).

```
>>> s = input('Enter something --> ')
Enter something --> Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
EOFError
```

Python raises an error called `EOFError` which basically means it found an *end of file* symbol (which is represented by `ctrl-d`) when it did not expect to see it.

Handling Exceptions

We can handle exceptions using the `try..except` statement. We basically put our usual statements within the try-block and put all our error handlers in the except-block.

Example (save as `exceptions_handle.py`):

```
try:
    text = input('Enter something --> ')
except EOFError:
    print('Why did you do an EOF on me?')
except KeyboardInterrupt:
    print('You cancelled the operation.')
else:
    print('You entered {}'.format(text))
```

Output:

```
# Press ctrl + d
$ python exceptions_handle.py
Enter something --> Why did you do an EOF on me?

# Press ctrl + c
$ python exceptions_handle.py
Enter something --> ^CYou cancelled the operation.

$ python exceptions_handle.py
Enter something --> No exceptions
You entered No exceptions
```

How It Works

We put all the statements that might raise exceptions/errors inside the `try` block and then put handlers for the appropriate errors/exceptions in the `except` clause/block. The `except` clause can handle a single specified error or exception, or a parenthesized list of errors/exceptions. If no names of errors or exceptions are supplied, it will handle *all* errors and exceptions.

Note that there has to be at least one `except` clause associated with every `try` clause. Otherwise, what's the point of having a try block?

If any error or exception is not handled, then the default Python handler is called which just stops the execution of the program and prints an error message. We have already seen this in action above.

You can also have an `else` clause associated with a `try..except` block. The `else` clause is executed if no exception occurs.

In the next example, we will also see how to get the exception object so that we can retrieve additional information.

Raising Exceptions

You can *raise* exceptions using the `raise` statement by providing the name of the error/exception and the exception object that is to be *thrown*.

The error or exception that you can raise should be a class which directly or indirectly must be a derived class of the `Exception` class.

Example (save as `exceptions_raise.py`):

```
class ShortInputException(Exception):
    '''A user-defined exception class.'''
    def __init__(self, length, atleast):
        Exception.__init__(self)
        self.length = length
        self.atleast = atleast

try:
    text = input('Enter something --> ')
    if len(text) < 3:
        raise ShortInputException(len(text), 3)
    # Other work can continue as usual here
except EOFError:
    print('Why did you do an EOF on me?')
except ShortInputException as ex:
    print(('ShortInputException: The input was ' +
           '{0} long, expected at least {1}')
          .format(ex.length, ex.atleast))
else:
    print('No exception was raised.')
```

Output:

```
$ python exceptions_raise.py
Enter something --> a
ShortInputException: The input was 1 long, expected at least 3

$ python exceptions_raise.py
Enter something --> abc
No exception was raised.
```

How It Works

Here, we are creating our own exception type. This new exception type is called `ShortInputException`. It has two fields - `length` which is the length of the given input, and `atleast` which is the minimum length that the program was expecting.

In the `except` clause, we mention the class of error which will be stored `as` the variable name to hold the corresponding error/exception object. This is analogous to parameters and arguments in a function call. Within this particular `except` clause, we use the `length` and `atleast` fields of the exception object to print an appropriate message to the user.

Try ... Finally

Suppose you are reading a file in your program. How do you ensure that the file object is closed properly whether or not an exception was raised? This can be done using the `finally` block.

Save this program as `exceptions_finally.py`:

```
import sys
import time

f = None
try:
    f = open("poem.txt")
    # Our usual file-reading idiom
    while True:
        line = f.readline()
        if len(line) == 0:
            break
        print(line, end='')
        sys.stdout.flush()
        print("Press ctrl+c now")
        # To make sure it runs for a while
        time.sleep(2)
except IOError:
    print("Could not find file poem.txt")
except KeyboardInterrupt:
    print("!! You cancelled the reading from the file.")
finally:
    if f:
        f.close()
    print("(Cleaning up: Closed the file)")
```

Output:

```
$ python exceptions_finally.py
Programming is fun
Press ctrl+c now
^C!! You cancelled the reading from the file.
(Cleaning up: Closed the file)
```

How It Works

We do the usual file-reading stuff, but we have arbitrarily introduced sleeping for 2 seconds after printing each line using the `time.sleep` function so that the program runs slowly (Python is very fast by nature). When the program is still running, press `ctrl + c` to interrupt/cancel the program.

Observe that the `KeyboardInterrupt` exception is thrown and the program quits. However, before the program exits, the finally clause is executed and the file object is always closed.

Notice that a variable assigned a value of 0 or `None` or a variable which is an empty sequence or collection is considered `False` by Python. This is why we can use `if: f` in the code above.

Also note that we use `sys.stdout.flush()` after `print` so that it prints to the screen immediately.

The with statement

Acquiring a resource in the `try` block and subsequently releasing the resource in the `finally` block is a common pattern. Hence, there is also a `with` statement that enables this to be done in a clean manner:

Save as `exceptions_using_with.py` :

```
with open("poem.txt") as f:
    for line in f:
        print(line, end='')
```

How It Works

The output should be same as the previous example. The difference here is that we are using the `open` function with the `with` statement - we leave the closing of the file to be done automatically by `with open` .

What happens behind the scenes is that there is a protocol used by the `with` statement. It fetches the object returned by the `open` statement, let's call it "thefile" in this case.

It *always* calls the `thefile.__enter__` function before starting the block of code under it and *always* calls `thefile.__exit__` after finishing the block of code.

So the code that we would have written in a `finally` block should be taken care of automatically by the `__exit__` method. This is what helps us to avoid having to use explicit `try..finally` statements repeatedly.

More discussion on this topic is beyond scope of this book, so please refer PEP 343 for a comprehensive explanation.

Summary

We have discussed the usage of the `try..except` and `try..finally` statements. We have seen how to create our own exception types and how to raise exceptions as well.

Next, we will explore the Python Standard Library.

Standard Library

The Python Standard Library contains a huge number of useful modules and is part of every standard Python installation. It is important to become familiar with the Python Standard Library since many problems can be solved quickly if you are familiar with the range of things that these libraries can do.

We will explore some of the commonly used modules in this library. You can find complete details for all of the modules in the Python Standard Library in the 'Library Reference' section of the documentation that comes with your Python installation.

Let us explore a few useful modules.

> CAUTION: If you find the topics in this chapter too advanced, you may skip this chapter. However, I highly recommend coming back to this chapter when you are more comfortable with programming using Python.

`sys` module

The `sys` module contains system-specific functionality. We have already seen that the `sys.argv` list contains the command-line arguments.

Suppose we want to check the version of the Python software being used, the `sys` module gives us that information.

```
>>> import sys
>>> sys.version_info
sys.version_info(major=3, minor=6, micro=0, releaselevel='final', serial=0)
>>> sys.version_info.major == 3
True
```

How It Works

The `sys` module has a `version_info` tuple that gives us the version information. The first entry is the major version. We can pull out this information to use it.

`logging` module

What if you wanted to have some debugging messages or important messages to be stored somewhere so that you can check whether your program has been running as you would expect it? How do you "store somewhere" these messages? This can be achieved using the `logging` module.

Save as `stdlib_logging.py` :

```
import os
import platform
import logging

if platform.platform().startswith('Windows'):
    logging_file = os.path.join(os.getenv('HOMEDRIVE'),
                                os.getenv('HOMEPATH'),
                                'test.log')
else:
    logging_file = os.path.join(os.getenv('HOME'),
                                'test.log')

print("Logging to", logging_file)

logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s : %(levelname)s : %(message)s',
    filename=logging_file,
    filemode='w',
)

logging.debug("Start of the program")
logging.info("Doing something")
logging.warning("Dying now")
```

Output:

```
$ python stdlib_logging.py
Logging to /Users/swa/test.log

$ cat /Users/swa/test.log
2014-03-29 09:27:36,660 : DEBUG : Start of the program
2014-03-29 09:27:36,660 : INFO : Doing something
2014-03-29 09:27:36,660 : WARNING : Dying now
```

The `cat` command is used in the command line to read the 'test.log' file. If the `cat` command is not available, you can open the `test.log` file in a text editor instead.

How It Works

We use three modules from the standard library - the `os` module for interacting with the operating system, the `platform` module for information about the platform i.e. the operating system and the `logging` module to *log* information.

First, we check which operating system we are using by checking the string returned by `platform.platform()` (for more information, see `import platform; help(platform)`). If it is Windows, we figure out the home drive, the home folder and the filename where we want to store the information. Putting these three parts together, we get the full location of the file. For other platforms, we need to know just the home folder of the user and we get the full location of the file.

We use the `os.path.join()` function to put these three parts of the location together. The reason to use a special function rather than just adding the strings together is because this function will ensure the full location matches the format expected by the operating system. Note: the `join()' method we use here that's part of the` os `module is different from the string method` join()` that we've used elsewhere in this book.

We configure the `logging` module to write all the messages in a particular format to the file we have specified.

Finally, we can put messages that are either meant for debugging, information, warning or even critical messages. Once the program has run, we can check this file and we will know what happened in the program, even though no information was displayed to the user running the program.

Module of the Week Series

There is much more to be explored in the standard library such as debugging, handling command line options, regular expressions and so on.

The best way to further explore the standard library is to read Doug Hellmann's excellent Python Module of the Week series (also available as a book) and reading the Python documentation.

Summary

We have explored some of the functionality of many modules in the Python Standard Library. It is highly recommended to browse through the Python Standard Library documentation to get an idea of all the modules that are available.

Next, we will cover various aspects of Python that will make our tour of Python more *complete*.

More

So far we have covered a majority of the various aspects of Python that you will use. In this chapter, we will cover some more aspects that will make our knowledge of Python more well-rounded.

Passing tuples around

Ever wished you could return two different values from a function? You can. All you have to do is use a tuple.

```
>>> def get_error_details():
...     return (2, 'details')
...
>>> errnum, errstr = get_error_details()
>>> errnum
2
>>> errstr
'details'
```

Notice that the usage of `a, b = <some expression>` interprets the result of the expression as a tuple with two values.

This also means the fastest way to swap two variables in Python is:

```
>>> a = 5; b = 8
>>> a, b
(5, 8)
>>> a, b = b, a
>>> a, b
(8, 5)
```

Special Methods

There are certain methods such as the `__init__` and `__del__` methods which have special significance in classes.

Special methods are used to mimic certain behaviors of built-in types. For example, if you want to use the `x[key]` indexing operation for your class (just like you use it for lists and tuples), then all you have to do is implement the `__getitem__()` method and your job is

done. If you think about it, this is what Python does for the `list` class itself!

Some useful special methods are listed in the following table. If you want to know about all the special methods, see the manual.

- `__init__(self, ...)`
  - This method is called just before the newly created object is returned for usage.
- `__del__(self)`
  - Called just before the object is destroyed (which has unpredictable timing, so avoid using this)
- `__str__(self)`
  - Called when we use the `print` function or when `str()` is used.
- `__lt__(self, other)`
  - Called when the *less than* operator (<) is used. Similarly, there are special methods for all the operators (+, >, etc.)
- `__getitem__(self, key)`
  - Called when `x[key]` indexing operation is used.
- `__len__(self)`
  - Called when the built-in `len()` function is used for the sequence object.

Single Statement Blocks

We have seen that each block of statements is set apart from the rest by its own indentation level. Well, there is one caveat. If your block of statements contains only one single statement, then you can specify it on the same line of, say, a conditional statement or looping statement. The following example should make this clear:

```
>>> flag = True
>>> if flag: print('Yes')
...
Yes
```

Notice that the single statement is used in-place and not as a separate block. Although, you can use this for making your program *smaller*, I strongly recommend avoiding this short-cut method, except for error checking, mainly because it will be much easier to add an extra statement if you are using proper indentation.

Lambda Forms

A `lambda` statement is used to create new function objects. Essentially, the `lambda` takes a parameter followed by a single expression. Lambda becomes the body of the function. The value of this expression is returned by the new function.

Example (save as `more_lambda.py`):

```
points = [{'x': 2, 'y': 3},
          {'x': 4, 'y': 1}]
points.sort(key=lambda i: i['y'])
print(points)
```

Output:

```
$ python more_lambda.py
[{'y': 1, 'x': 4}, {'y': 3, 'x': 2}]
```

How It Works

Notice that the `sort` method of a `list` can take a `key` parameter which determines how the list is sorted (usually we know only about ascending or descending order). In our case, we want to do a custom sort, and for that we need to write a function. Instead of writing a separate `def` block for a function that will get used in only this one place, we use a lambda expression to create a new function.

List Comprehension

List comprehensions are used to derive a new list from an existing list. Suppose you have a list of numbers and you want to get a corresponding list with all the numbers multiplied by 2 only when the number itself is greater than 2. List comprehensions are ideal for such situations.

Example (save as `more_list_comprehension.py`):

```
listone = [2, 3, 4]
listtwo = [2*i for i in listone if i > 2]
print(listtwo)
```

Output:

```
$ python more_list_comprehension.py
[6, 8]
```

How It Works

Here, we derive a new list by specifying the manipulation to be done (`2*i`) when some condition is satisfied (`if i > 2`). Note that the original list remains unmodified.

The advantage of using list comprehensions is that it reduces the amount of boilerplate code required when we use loops to process each element of a list and store it in a new list.

Receiving Tuples and Dictionaries in Functions

There is a special way of receiving parameters to a function as a tuple or a dictionary using the `*` or `**` prefix respectively. This is useful when taking variable number of arguments in the function.

```
>>> def powersum(power, *args):
...     '''Return the sum of each argument raised to the specified power.'''
...     total = 0
...     for i in args:
...         total += pow(i, power)
...     return total
...
>>> powersum(2, 3, 4)
25
>>> powersum(2, 10)
100
```

Because we have a `*` prefix on the `args` variable, all extra arguments passed to the function are stored in `args` as a tuple. If a `**` prefix had been used instead, the extra parameters would be considered to be key/value pairs of a dictionary.

The assert statement

The `assert` statement is used to assert that something is true. For example, if you are very sure that you will have at least one element in a list you are using and want to check this, and raise an error if it is not true, then `assert` statement is ideal in this situation. When the assert statement fails, an `AssertionError` is raised. The `pop()` method removes and returns the last item from the list.

```
>>> mylist = ['item']
>>> assert len(mylist) >= 1
>>> mylist.pop()
'item'
>>> assert len(mylist) >= 1
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

The `assert` statement should be used judiciously. Most of the time, it is better to catch exceptions, either handle the problem or display an error message to the user and then quit.

Decorators

Decorators are a shortcut to applying wrapper functions. This is helpful to "wrap" functionality with the same code over and over again. For example, I created a `retry` decorator for myself that I can just apply to any function and if any exception is thrown during a run, it is retried again, till a maximum of 5 times and with a delay between each retry. This is especially useful for situations where you are trying to make a network call to a remote computer:

```python
from time import sleep
from functools import wraps
import logging
logging.basicConfig()
log = logging.getLogger("retry")


def retry(f):
    @wraps(f)
    def wrapper_function(*args, **kwargs):
        MAX_ATTEMPTS = 5
        for attempt in range(1, MAX_ATTEMPTS + 1):
            try:
                return f(*args, **kwargs)
            except Exception:
                log.exception("Attempt %s/%s failed : %s",
                              attempt,
                              MAX_ATTEMPTS,
                              (args, kwargs))
                sleep(10 * attempt)
        log.critical("All %s attempts failed : %s",
                     MAX_ATTEMPTS,
                     (args, kwargs))
    return wrapper_function


counter = 0


@retry
def save_to_database(arg):
    print("Write to a database or make a network call or etc.")
    print("This will be automatically retried if exception is thrown.")
    global counter
    counter += 1
    # This will throw an exception in the first call
    # And will work fine in the second call (i.e. a retry)
    if counter < 2:
        raise ValueError(arg)


if __name__ == '__main__':
    save_to_database("Some bad value")
```

Output:

```
$ python more_decorator.py
Write to a database or make a network call or etc.
This will be automatically retried if exception is thrown.
ERROR:retry:Attempt 1/5 failed : (('Some bad value',), {})
Traceback (most recent call last):
  File "more_decorator.py", line 14, in wrapper_function
    return f(*args, **kwargs)
  File "more_decorator.py", line 39, in save_to_database
    raise ValueError(arg)
ValueError: Some bad value
Write to a database or make a network call or etc.
This will be automatically retried if exception is thrown.
```

How It Works

See:

- http://www.ibm.com/developerworks/linux/library/l-cpdecor.html
- http://toumorokoshi.github.io/dry-principles-through-python-decorators.html

Differences between Python 2 and Python 3

See:

- "Six" library
- Porting to Python 3 Redux by Armin
- Python 3 experience by PyDanny
- Official Django Guide to Porting to Python 3
- Discussion on What are the advantages to python 3.x?

Summary

We have covered some more features of Python in this chapter and yet we haven't covered all the features of Python. However, at this stage, we have covered most of what you are ever going to use in practice. This is sufficient for you to get started with whatever programs you are going to create.

Next, we will discuss how to explore Python further.

What Next

If you have read this book thoroughly till now and practiced writing a lot of programs, then you must have become comfortable and familiar with Python. You have probably created some Python programs to try out stuff and to exercise your Python skills as well. If you have not done it already, you should. The question now is 'What Next?'.

I would suggest that you tackle this problem:

> Create your own command-line *address-book* program using which you can browse, add, modify, delete or search for your contacts such as friends, family and colleagues and their information such as email address and/or phone number. Details must be stored for later retrieval.

This is fairly easy if you think about it in terms of all the various stuff that we have come across till now. If you still want directions on how to proceed, then here's a hint [1].

Once you are able to do this, you can claim to be a Python programmer. Now, immediately send me an email thanking me for this great book ;-). This step is optional but recommended. Also, please consider buying a printed copy to support the continued development of this book.

If you found that program easy, here's another one:

> Implement the replace command. This command will replace one string with another in the list of files provided.

The replace command can be as simple or as sophisticated as you wish, from simple string substitution to looking for patterns (regular expressions).

Next Projects

If you found above programs easy to create, then look at this comprehensive list of projects and try writing your own programs: https://github.com/thekarangoel/Projects#numbers (the same list is also at Martyr2's Mega Project List).

Also see:

- Exercises for Programmers: 57 Challenges to Develop Your Coding Skills
- Intermediate Python Projects.

Example Code

The best way to learn a programming language is to write a lot of code and read a lot of code:

- Python Cookbook is an extremely valuable collection of recipes or tips on how to solve certain kinds of problems using Python. This is a must-read for every Python user.
- Python Module of the Week is another excellent must-read guide to the Standard Library.

Advice

- The Hitchhiker's Guide to Python!
- The Elements of Python Style
- Python Big Picture
- "Writing Idiomatic Python" ebook (paid)

Videos

- Full Stack Web Development with Flask
- PyVideo

Questions and Answers

- Official Python Dos and Don'ts
- Official Python FAQ
- Norvig's list of Infrequently Asked Questions
- Python Interview Q & A
- StackOverflow questions tagged with python

Tutorials

- Hidden features of Python
- What's the one code snippet/python trick/etc did you wish you knew when you learned python?
- Awaretek's comprehensive list of Python tutorials

Discussion

If you are stuck with a Python problem, and don't know whom to ask, then the python-tutor list is the best place to ask your question.

Make sure you do your homework by trying to solving the problem yourself first and ask smart questions.

News

If you want to learn what is the latest in the world of Python, then follow the Official Python Planet.

Installing libraries

There are a huge number of open source libraries at the Python Package Index which you can use in your own programs.

To install and use these libraries, you can use pip.

Creating a Website

Learn Flask to create your own website. Some resources to get started:

- Flask Official Quickstart
- The Flask Mega-Tutorial
- Example Flask Projects

Graphical Software

Suppose you want to create your own graphical programs using Python. This can be done using a GUI (Graphical User Interface) library with their Python bindings. Bindings are what allow you to write programs in Python and use the libraries which are themselves written in C or C++ or other languages.

There are lots of choices for GUI using Python:

- Kivy
    - http://kivy.org

- PyGTK
    - This is the Python binding for the GTK+ toolkit which is the foundation upon which GNOME is built. GTK+ has many quirks in usage but once you become comfortable, you can create GUI apps fast. The Glade graphical interface designer is indispensable. The documentation is yet to improve. GTK+ works well on GNU/Linux but its port to Windows is incomplete. You can create both free as well as proprietary software using GTK+. To get started, read the PyGTK tutorial.
- PyQt
    - This is the Python binding for the Qt toolkit which is the foundation upon which the KDE is built. Qt is extremely easy to use and very powerful especially due to the Qt Designer and the amazing Qt documentation. PyQt is free if you want to create open source (GPL'ed) software and you need to buy it if you want to create proprietary closed source software. Starting with Qt 4.5 you can use it to create non-GPL software as well. To get started, read about PySide.
- wxPython
    - This is the Python bindings for the wxWidgets toolkit. wxPython has a learning curve associated with it. However, it is very portable and runs on GNU/Linux, Windows, Mac and even embedded platforms. There are many IDEs available for wxPython which include GUI designers as well such as SPE (Stani's Python Editor) and the wxGlade GUI builder. You can create free as well as proprietary software using wxPython. To get started, read the wxPython tutorial.

Summary of GUI Tools

For more choices, see the GuiProgramming wiki page at the official python website.

Unfortunately, there is no one standard GUI tool for Python. I suggest that you choose one of the above tools depending on your situation. The first factor is whether you are willing to pay to use any of the GUI tools. The second factor is whether you want the program to run only on Windows or on Mac and GNU/Linux or all of them. The third factor, if GNU/Linux is a chosen platform, is whether you are a KDE or GNOME user on GNU/Linux.

For a more detailed and comprehensive analysis, see Page 26 of the 'The Python Papers, Volume 3, Issue 1' (PDF).

Various Implementations

There are usually two parts a programming language - the language and the software. A language is *how* you write something. The software is *what* actually runs our programs.

We have been using the *CPython* software to run our programs. It is referred to as CPython because it is written in the C language and is the *Classical Python interpreter*.

There are also other software that can run your Python programs:

- Jython
    - A Python implementation that runs on the Java platform. This means you can use Java libraries and classes from within Python language and vice-versa.
- IronPython
    - A Python implementation that runs on the .NET platform. This means you can use .NET libraries and classes from within Python language and vice-versa.
- PyPy
    - A Python implementation written in Python! This is a research project to make it fast and easy to improve the interpreter since the interpreter itself is written in a dynamic language (as opposed to static languages such as C, Java or C# in the above three implementations)

There are also others such as CLPython - a Python implementation written in Common Lisp and Brython which is an implementation on top of a JavaScript interpreter which could mean that you can use Python (instead of JavaScript) to write your web-browser ("Ajax") programs.

Each of these implementations have their specialized areas where they are useful.

Functional Programming (for advanced readers)

When you start writing larger programs, you should definitely learn more about a functional approach to programming as opposed to the class-based approach to programming that we learned in the object-oriented programming chapter:

- Functional Programming Howto by A.M. Kuchling
- Functional programming chapter in 'Dive Into Python' book
- Functional Programming with Python presentation
- Funcy library
- PyToolz library

Summary

We have now come to the end of this book but, as they say, this is the *the beginning of the end*!. You are now an avid Python user and you are no doubt ready to solve many problems using Python. You can start automating your computer to do all kinds of previously unimaginable things or write your own games and much much more. So, get started!

> [1]. Create a class to represent the person's information. Use a dictionary to store person objects with their name as the key. Use the pickle module to store the objects persistently on your hard disk. Use the dictionary built-in methods to add, delete and modify the persons. ↩

Appendix: FLOSS

> NOTE: Please note that this section was written in 2003, so some of this will sound quaint to you :-)

"Free/Libre and Open Source Software", in short, FLOSS is based on the concept of a community, which itself is based on the concept of sharing, and particularly the sharing of knowledge. FLOSS are free for usage, modification and redistribution.

If you have already read this book, then you are already familiar with FLOSS since you have been using *Python* all along and Python is an open source software!

Here are some examples of FLOSS to give an idea of the kind of things that community sharing and building can create:

Linux: This is a FLOSS OS kernel used in the GNU/Linux operating system. Linux, the kernel, was started by Linus Torvalds as a student. Android is based on Linux. Any website you use these days will mostly be running on Linux.

Ubuntu: This is a community-driven distribution, sponsored by Canonical and it is the most popular GNU/Linux distribution today. It allows you to install a plethora of FLOSS available and all this in an easy-to-use and easy-to-install manner. Best of all, you can just reboot your computer and run GNU/Linux off the CD! This allows you to completely try out the new OS before installing it on your computer. However, Ubuntu is not entirely free software; it contains proprietary drivers, firmware, and applications.

LibreOffice: This is an excellent community-driven and developed office suite with a writer, presentation, spreadsheet and drawing components among other things. It can even open and edit MS Word and MS PowerPoint files with ease. It runs on almost all platforms and is entirely free, libre and open source software.

Mozilla Firefox: This is *the* best web browser. It is blazingly fast and has gained critical acclaim for its sensible and impressive features. The extensions concept allows any kind of plugins to be used.

Mono: This is an open source implementation of the Microsoft .NET platform. It allows .NET applications to be created and run on GNU/Linux, Windows, FreeBSD, Mac OS and many other platforms as well.

Apache web server: This is the popular open source web server. In fact, it is *the* most popular web server on the planet! It runs nearly more than half of the websites out there. Yes, that's right - Apache handles more websites than all the competition (including

Microsoft IIS) combined.

VLC Player: This is a video player that can play anything from DivX to MP3 to Ogg to VCDs and DVDs to ... who says open source ain't fun? ;-)

This list is just intended to give you a brief idea - there are many more excellent FLOSS out there, such as the Perl language, PHP language, Drupal content management system for websites, PostgreSQL database server, TORCS racing game, KDevelop IDE, Xine - the movie player, VIM editor, Quanta+ editor, Banshee audio player, GIMP image editing program, ... This list could go on forever.

To get the latest buzz in the FLOSS world, check out the following websites:

- OMG! Ubuntu!
- Web Upd8
- DistroWatch
- Planet Debian

Visit the following websites for more information on FLOSS:

- GitHub Explore
- Code Triage
- SourceForge
- FreshMeat

So, go ahead and explore the vast, free and open world of FLOSS!

Appendix: Colophon

Almost all of the software that I have used in the creation of this book are FLOSS.

Birth of the Book

In the first draft of this book, I had used Red Hat 9.0 Linux as the foundation of my setup and in the sixth draft, I used Fedora Core 3 Linux as the basis of my setup.

Initially, I was using KWord to write the book (as explained in the history lesson).

Teenage Years

Later, I switched to DocBook XML using Kate but I found it too tedious. So, I switched to OpenOffice which was just excellent with the level of control it provided for formatting as well as the PDF generation, but it produced very sloppy HTML from the document.

Finally, I discovered XEmacs and I rewrote the book from scratch in DocBook XML (again) after I decided that this format was the long term solution.

In the sixth draft, I decided to use Quanta+ to do all the editing. The standard XSL stylesheets that came with Fedora Core 3 Linux were being used. However, I had written a CSS document to give color and style to the HTML pages. I had also written a crude lexical analyzer, in Python of course, which automatically provides syntax highlighting to all the program listings.

For the seventh draft, I was using MediaWiki as the basis of my setup. I used to edit everything online and the readers can directly read/edit/discuss within the wiki website, but I ended up spending more time fighting spam than writing.

For the eight draft, I used Vim, Pandoc, and Mac OS X.

For the ninth draft, I switched to AsciiDoc format and used Emacs 24.3, tomorrow theme, Fira Mono font and adoc-mode to write.

Now

2016: I got tired of several minor rendering issues in AsciiDoctor, like the `++` in `C/C++` would disappear and it was hard to keep track of escaping such minor things. Plus, I had become reluctant to edit the text because of the complex Asciidoc format.

For the tenth draft, I switched to writing in Markdown + GitBook format, using the Spacemacs editor.

About the Author

See https://www.swaroopch.com/about/

Appendix: History Lesson

I first started with Python when I needed to write an installer for software I had written called 'Diamond' so that I could make the installation easy. I had to choose between Python and Perl bindings for the Qt library. I did some research on the web and I came across an article by Eric S. Raymond, a famous and respected hacker, where he talked about how Python had become his favorite programming language. I also found out that the PyQt bindings were more mature compared to Perl-Qt. So, I decided that Python was the language for me.

Then, I started searching for a good book on Python. I couldn't find any! I did find some O'Reilly books but they were either too expensive or were more like a reference manual than a guide. So, I settled for the documentation that came with Python. However, it was too brief and small. It did give a good idea about Python but was not complete. I managed with it since I had previous programming experience, but it was unsuitable for newbies.

About six months after my first brush with Python, I installed the (then) latest Red Hat 9.0 Linux and I was playing around with KWord. I got excited about it and suddenly got the idea of writing some stuff on Python. I started writing a few pages but it quickly became 30 pages long. Then, I became serious about making it more useful in a book form. After a *lot* of rewrites, it has reached a stage where it has become a useful guide to learning the Python language. I consider this book to be my contribution and tribute to the open source community.

This book started out as my personal notes on Python and I still consider it in the same way, although I've taken a lot of effort to make it more palatable to others :)

In the true spirit of open source, I have received lots of constructive suggestions, criticisms and feedback from enthusiastic readers which has helped me improve this book a lot.

Status Of The Book

The book needs the help of its readers such as yourselves to point out any parts of the book which are not good, not comprehensible or are simply wrong. Please write to the main author or the respective translators with your comments and suggestions.

Appendix: Revision History

- 4.0

  - 19 Jan 2016
  - Switched back to Python 3
  - Switched back to Markdown, using GitBook and Spacemacs
- 3.0
  - 31 Mar 2014
  - Rewritten for Python 2 using AsciiDoc and adoc-mode.
- 2.1
  - 03 Aug 2013
  - Rewritten using Markdown and Jason Blevins' Markdown Mode
- 2.0
  - 20 Oct 2012
  - Rewritten in Pandoc format, thanks to my wife who did most of the conversion from the Mediawiki format
  - Simplifying text, removing non-essential sections such as `nonlocal` and metaclasses
- 1.90
  - 04 Sep 2008 and still in progress
  - Revival after a gap of 3.5 years!
  - Rewriting for Python 3.0
  - Rewrite using http://www.mediawiki.org[MediaWiki] (again)
- 1.20
  - 13 Jan 2005
  - Complete rewrite using Quanta+ on Fedora Core 3 with lot of corrections and updates. Many new examples. Rewrote my DocBook setup from scratch.
- 1.15
  - 28 Mar 2004
  - Minor revisions
- 1.12
  - 16 Mar 2004
  - Additions and corrections
- 1.10
  - 09 Mar 2004
  - More typo corrections, thanks to many enthusiastic and helpful readers.
- 1.00
  - 08 Mar 2004

    - After tremendous feedback and suggestions from readers, I have made significant revisions to the content along with typo corrections.
- 0.99
    - 22 Feb 2004
    - Added a new chapter on modules. Added details about variable number of arguments in functions.
- 0.98
    - 16 Feb 2004
    - Wrote a Python script and CSS stylesheet to improve XHTML output, including a crude-yet-functional lexical analyzer for automatic VIM-like syntax highlighting of the program listings.
- 0.97
    - 13 Feb 2004
    - Another completely rewritten draft, in DocBook XML (again). Book has improved a lot - it is more coherent and readable.
- 0.93
    - 25 Jan 2004
    - Added IDLE talk and more Windows-specific stuff
- 0.92
    - 05 Jan 2004
    - Changes to few examples.
- 0.91
    - 30 Dec 2003
    - Corrected typos. Improvised many topics.
- 0.90
    - 18 Dec 2003
    - Added 2 more chapters. OpenOffice format with revisions.
- 0.60
    - 21 Nov 2003
    - Fully rewritten and expanded.
- 0.20
    - 20 Nov 2003
    - Corrected some typos and errors.
- 0.15
    - 20 Nov 2003

    - Converted to DocBook XML with XEmacs.
- 0.10
    - 14 Nov 2003
    - Initial draft using KWord.

Translations

There are many translations of the book available in different human languages, thanks to many tireless volunteers!

If you want to help with these translations, please see the list of volunteers and languages below and decide if you want to start a new translation or help in existing translation projects.

If you plan to start a new translation, please read the Translation how-to.

Arabic

Below is the link for the Arabic version. Thanks to Ashraf Ali Khalaf for translating the book, you can read the whole book online at http://www.khaledhosny.org/byte-of-python/index.html or you can download it from sourceforge.net for more info see http://itwadi.com/byteofpython_arabi.

Azerbaijani

Jahangir Shabiyev (c.shabiev@gmail.com) has volunteered to translate the book to Azerbaijani. The translation is in progress at https://www.gitbook.com/book/jahangir-sh/piton-sancmasi

Brazilian Portuguese

There are two translations in various levels of completion and accessibility. The older translation is now missing/lost, and newer translation is incomplete.

Samuel Dias Neto (samuel.arataca@gmail.com) made the first Brazilian Portuguese translation (older translation) of this book when Python was in 2.3.5 version. This is no longer publicly accessible.

Rodrigo Amaral (rodrigoamaral@gmail.com) has volunteered to translate the book to Brazilian Portuguese, (newer translation) which still remains to be completed.

Catalan

Moises Gomez (moisesgomezgiron@gmail.com) has volunteered to translate the book to Catalan. The translation is in progress.

> Moisès Gómez - I am a developer and also a teacher of programming (normally for people without any previous experience).
>
> Some time ago I needed to learn how to program in Python, and Swaroop's work was really helpful. Clear, concise, and complete enough. Just what I needed.
>
> After this experience, I thought some other people in my country could take benefit from it too. But English language can be a barrier.
>
> So, why not try to translate it? And I did for a previous version of BoP.
>
> I my country there are two official languages. I selected the Catalan language assuming that others will translate it to the more widespread Spanish.

Chinese

In 2017 which is after 11 years, Mo Lun (i@molun.net) re-translated the book from the beginning based on Version 4.0. And the translation is storaged in GitHub and Gitbook. He is keeping follow this translated edition and ready to fix it if there is any wrong or mistake in the translated BoP.

The 2017 translation edition is available in https://bop.molun.net.

Mo Lun Says:

> I am a common journalism student from CYU, Beijing. And actually, I am an absolute newbie in Python programming when I start to translate this book. Initially, it was just a whim, but when I done this work, I realized that a decision triggered by interest had prompted me to go so far.
>
> With the help of my predecessors' translations and the vast amount of information provided by the developed Internet, and with the help of my friends, I prudently presented this translation edition. I just hope my translation work will help other newcomers in learning Python.
>
> At the same time, I am always waiting for my translation of the comments and suggestions, and ready to change or improve this superficial work.

Earlier Chinese translation

In 2005, Shen Jieyuan translated this book with version 1.20 to Chinese and published it to the Internet. This is the first Chinese edition. In BoP official site, he was called Juan Shen, with Email adderss orion_val@163.com。This edition has been disseminated on the network widely, and the links provided by BoP official site are not available anymore, so that its original source is unable to find. Therefore in here can't provide a certain address. But you can try to search keywords like "简明Python教程 沈洁元" to find a copy.

Juan Shen says:

> I am a postgraduate at Wireless Telecommunication Graduate School, Beijing University of Technology, China PR. My current research interest is on the synchronization, channel estimation and multi-user detection of multicarrier CDMA system. Python is my major programming language for daily simulation and research job, with the help of Python Numeric, actually. I learned Python just half a year before, but as you can see, it's really easy-understanding, easy-to-use and productive. Just as what is ensured in Swaroop's book, 'It's my favorite programming language now'.
>
> 'A Byte of Python' is my tutorial to learn Python. It's clear and effective to lead you into a world of Python in the shortest time. It's not too long, but efficiently covers almost all important things in Python. I think 'A Byte of Python' should be strongly recommendable for newbies as their first Python tutorial. Just dedicate my translation to the potential millions of Python users in China.

Chinese Traditional

Fred Lin (gasolin@gmail.com) has volunteered to translate the book to Chinese Traditional.

It is available at http://code.google.com/p/zhpy/wiki/ByteOfZhpy.

An exciting feature of this translation is that it also contains the *executable chinese python sources* side by side with the original python sources.

> Fred Lin - I'm working as a network firmware engineer at Delta Network, and I'm also a contributor of TurboGears web framework.
>
> As a python evangelist (:-p), I need some material to promote python language. I found 'A Byte of Python' hit the sweet point for both newbies and experienced programmers. 'A Byte of Python' elaborates the python essentials with affordable size.
>
> The translation are originally based on simplified chinese version, and soon a lot of rewrite were made to fit the current wiki version and the quality of reading.
>
> The recent chinese traditional version also featured with executable chinese python sources, which are achieved by my new 'zhpy' (python in chinese) project (launch from Aug 07).
>
> zhpy(pronounce (Z.H.?, or zippy) build a layer upon python to translate or interact with python in chinese(Traditional or Simplified). This project is mainly aimed for education.

French

Gregory (coulix@ozforces.com.au) has volunteered to translate the book to French.

Gérard Labadie (gerard.labadie@gmail.com) has completed to translate the book to French.

German

Lutz Horn (lutz.horn@gmx.de), Bernd Hengelein (bernd.hengelein@gmail.com) and Christoph Zwerschke (cito@online.de) have volunteered to translate the book to German.

Translation can be found at http://cito.github.io/byte_of_python/

Lutz Horn says:

> I'm 32 years old and have a degree of Mathematics from University of Heidelberg, Germany. Currently I'm working as a software engineer on a publicly funded project to build a web portal for all things related to computer science in Germany.The main language I use as a professional is Java, but I try to do as much as possible with Python behind the scenes. Especially text analysis and conversion is very easy with Python. I'm not very familiar with GUI toolkits, since most of my programming is about web applications, where the user interface is build using Java frameworks like Struts. Currently I try to make more use of the functional programming features of Python and of generators. After taking a short look into Ruby, I was very impressed with the use of blocks in this language. Generally I like the dynamic nature of languages like Python and Ruby since it allows me to do things not possible in more static languages like Java.I've searched for some kind of introduction to programming, suitable to teach a complete non-programmer. I've found the book 'How to Think Like a Computer Scientist: Learning with Python', and 'Dive into Python'. The first is good for beginners but to long to translate. The second is not suitable for beginners. I think 'A Byte of Python' falls nicely between these, since it is not too long, written to the point, and at the same time verbose enough to teach a newbie. Besides this, I like the simple DocBook structure, which makes translating the text a generation the output in various formats a charm.

Bernd Hengelein says:

> Lutz and me are going to do the german translation together. We just started with the intro and preface but we will keep you informed about the progress we make. Ok, now some personal things about me. I am 34 years old and playing with computers since the 1980's, when the "Commodore C64" ruled the nurseries. After studying computer science I started working as a software engineer. Currently I am working in the field of medical imaging for a major german company. Although C++ is the main language I (have to) use for my daily work, I am constantly looking for new things to learn.Last year I fell in love with Python, which is a wonderful language, both for its possibilities and its beauty. I read somewhere in the net about a guy who said that he likes python, because the code looks so beautiful. In my opinion he's absolutly right. At the time I decided to learn python, I noticed that there is very little good documentation in german available. When I came across your book the spontaneous idea of a german translation crossed my mind. Luckily, Lutz had the same idea and we can now divide the work.I am looking forward to a good cooperation!

Greek

The Greek Ubuntu Community translated the book in Greek, for use in our on-line asynchronous Python lessons that take place in our forums. Contact @savvasradevic for more information.

Indonesian

Daniel (daniel.mirror@gmail.com) is translating the book to Indonesian at http://python.or.id/moin.cgi/ByteofPython.

Wisnu Priyambodo (cibermen@gmail.com) also has volunteered to translate the book to Indonesian.

Also, Bagus Aji Santoso (baguzzzaji@gmail.com) has volunteered.

Italian (first)

Enrico Morelli (mr.mlucci@gmail.com) and Massimo Lucci (morelli@cerm.unifi.it) have volunteered to translate the book to Italian.

The Italian translation is present at http://www.gentoo.it/Programmazione/byteofpython.

> *Massimo Lucci and Enrico Morelli* - we are working at the University of Florence (Italy) - Chemistry Department. I (Massimo) as service engineer and system administrator for Nuclear Magnetic Resonance Spectrometers; Enrico as service engineer and system administrator for our CED and parallel / clustered systems. We are programming on python since about seven years, we had experience working with Linux platforms since ten years. In Italy we are responsible and administrator for www.gentoo.it web site for Gentoo/Linux distrubution and www.nmr.it (now under construction) for Nuclear Magnetic Resonance applications and Congress Organization and Managements.That's all! We are impressed by the smart language used on your Book and we think this is essential for approaching the Python to new users (we are thinking about hundred of students and researcher working on our labs).

Italian (second)

An Italian translation has been created by Calvina Bice & colleagues at http://besthcgdropswebsite.com/translate/a-byte-of-python/.

Japanese

Shunro Dozono (dozono@gmail.com) is translating the book to Japanese.

Korean

Jeongbin Park (pjb7687@gmail.com) has translated the book to Korean - https://github.com/pjb7687/byte_of_python

> I am Jeongbin Park, currently working as a Biophysics & Bioinformatics researcher in Korea.
>
> A year ago, I was looking for a good tutorial/guide for Python to introduce it to my colleagues, because using Python in such research fields is becoming inevitable due to the user base is growing more and more.
>
> But at that time only few Python books are available in Korean, so I decided to translate your ebook because it looks like one of the best guides that I have ever read!
>
> Currently, the book is almost completely translated in Korean, except some of the text in introduction chapter and the appendixes.
>
> Thank you again for writing such a good guide!

Mongolian

Ariunsanaa Tunjin (luftballons2010@gmail.com) has volunteered to translate the book to Mongolian.

Update on Nov 22, 2009 : Ariunsanaa is on the verge of completing the translation.

Norwegian (bokmål)

Eirik Vågeskar is a high school student at Sandvika videregående skole in Norway, a blogger and currently translating the book to Norwegian (bokmål).

> *Eirik Vågeskar*: I have always wanted to program, but because I speak a small language, the learning process was much harder. Most tutorials and books are written in very technical English, so most high school graduates will not even have the vocabulary to understand what the tutorial is about. When I discovered this book, all my problems were solved. "A Byte of Python" used simple non-technical language to explain a programming language that is just as simple, and these two things make learning Python fun. After reading half of the book, I decided that the book was worth translating. I hope the translation will help people who have found themself in the same situation as me (especially young people), and maybe help spread interest for the language among people with less technical knowledge.

Polish

Dominik Kozaczko (dominik@kozaczko.info) has volunteered to translate the book to Polish. Translation is in progress and it's main page is available here: Ukąś Pythona.

Update : The translation is complete and ready as of Oct 2, 2009. Thanks to Dominik, his two students and their friend for their time and effort!

> *Dominik Kozaczko* - I'm a Computer Science and Information Technology teacher.

Portuguese

Artur Weber (arturweberguimaraes@gmail.com) has completed a translation of this book to Portuguese (as of Feb 21, 2018) at https://www.homeyou.com/~edu/introducao.

> *Artur Weber*: My students study at the Polytechnic faculty in Ecological University in the city of Curitiba (Brazil) and some of them are interested in different papers.
>
> As they are writing course and academic papers, they always look for interesting articles and pages. I also do my best to find interesting materials which can be the sources for their university works.
>
> I found the materials from your site useful for some of my students who are writing papers based on programming on Python. Actually, that is why I made a decision to perform a Portuguese translation to let my students who don't know English read exciting articles in their mother tongue (in Portuguese).

Romanian

Paul-Sebastian Manole (brokenthorn@gmail.com) has volunteered to translate this book to Romanian.

> *Paul-Sebastian Manole* - I'm a second year Computer Science student at Spiru Haret University, here in Romania. I'm more of a self-taught programmer and decided to learn a new language, Python. The web told me there was no better way to do so but read "A Byte of Python". That's how popular this book is (congratulations to the author for writing such an easy to read book). I started liking Python so I decided to help translate the latest version of Swaroop's book in Romanian. Although I could be the one with the first initiative, I'm just one volunteer so if you can help, please join me.

Russian

Vladimir Smolyar (v_2e@ukr.net) has completed a Russian translation at http://wombat.org.ua/AByteOfPython/.

Ukranian

Averkiev Andrey (averkiyev@ukr.net) has volunteered to translate the book to Russian, and perhaps Ukranian (time permitting).

Serbian

"BugSpice" (amortizerka@gmail.com) has completed a Serbian translation:

> This download link is no longer accessible.

More details at http://forum.ubuntu-rs.org/Thread-zagrljaj-pitona.

Slovak

Albertio Ward (albertioward@gmail.com) has translated the book to Slovak at http://www.fatcow.com/edu/python-swaroopch-sl/ :

> We are a non-profit organization called "Translation for education". We represent a group of people, mainly students and professors, of the Slavonic University. Here are students from different departments: linguistics, chemistry, biology, etc. We try to find interesting publications on the Internet that can be relevant for us and our university colleagues. Sometimes we find articles by ourselves; other times our professors help us choose the material for translation. After obtaining permission from authors we translate articles and post them in our blog which is available and accessible to our colleagues and friends. These translated publications often help students in their daily study routine.

Spanish

Alfonso de la Guarda Reyes (alfonsodg@ictechperu.net), Gustavo Echeverria (gustavo.echeverria@gmail.com), David Crespo Arroyo (davidcrespoarroyo@hotmail.com) and Cristian Bermudez Serna (crisbermud@hotmail.com) have volunteered to translate the book to Spanish.

Gustavo Echeverria says:

> I work as a software engineer in Argentina. I use mostly C# and .Net technologies at work but strictly Python or Ruby in my personal projects. I knew Python many years ago and I got stuck inmediately. Not so long after knowing Python I discovered this book and it helped me to learn the language. Then I volunteered to translate the book to Spanish. Now, after receiving some requests, I've begun to translate "A Byte of Python" with the help of Maximiliano Soler.

Cristian Bermudez Serna says:

> I am student of Telecommunications engineering at the University of Antioquia (Colombia). Months ago, i started to learn Python and found this wonderful book, so i volunteered to get the Spanish translation.

Swedish

Mikael Jacobsson (leochingkwake@gmail.com) has volunteered to translate the book to Swedish.

Turkish

Türker SEZER (tsezer@btturk.net) and Bugra Cakir (bugracakir@gmail.com) have volunteered to translate the book to Turkish. "Where is Turkish version? Bitse de okusak."

Translation How-to

1. The full source of the book is available from https://github.com/swaroopch/byte-of-python.
2. Please fork the repository.
3. Then, fetch the repository to your computer. You need to know how to use Git to do that.
4. Read the GitBook documentation, esp. the Markdown section.
5. Start editing the `.md` files to translate to your local language.
6. Sign up on GitBook.com, create a book and you can see a beautifully rendered website, with links to download PDF, EPUB, etc.

Feedback

The book needs the help of its readers such as yourselves to point out any parts of the book which are not good, not comprehensible or are simply wrong. Please write to the main author or the respective translators with your comments and suggestions.